

Taller de Laravel v.11 (I)

Fco Javier Tárrega (febrero 2025)

PREPARANDO EL ENTORNO DE DESARROLLO

En primer lugar, vamos a actualizar los repositorios de Ubuntu y a actualizar paquetes, abriremos la terminal de escritorio y en ella ejecutamos:

```
sudo apt update
sudo apt-get upgrade
sudo apt-get dist-upgrade
sudo apt-get autoremove
```

Laravel tiene algunos requisitos del sistema. Debes asegurarte de que tu servidor web tenga la siguiente versión mínima de PHP y extensiones:

- ✓ PHP >= 8.2
- ✓ Ctype PHP Extension
- ✓ cURL PHP Extension
- ✓ DOM PHP Extension
- ✓ Fileinfo PHP Extension
- ✓ Filter PHP Extension
- ✓ Hash PHP Extension
- ✓ Mbstring PHP Extension
- ✓ OpenSSL PHP Extension
- ✓ PCRE PHP Extension
- ✓ PDO PHP Extension
- ✓ Session PHP Extension
- ✓ Tokenizer PHP Extension
- ✓ XML PHP Extension

Teniendo en cuenta los requisitos de Laravel, instalamos Apache2, PHP, MariaDB y las extensiones con:

```
sudo apt install apache2 php libapache2-mod-php php-mbstring php-xmlrpc php-soap php-gd php-xml php-cli php-zip
php-bcmath php-tokenizer php-json php-pear mariadb-server php-mysql php-curl
```

Tras esta instalación, podemos comprobar que tenemos todos los requisitos creando y accediendo a un archivo en `/var/www/html/phpinfo.php` que contenga:

```
<?php
phpinfo();
?>
```

PREPARANDO EL ENTORNO DE DESARROLLO

Ahora ejecutamos el siguiente comando para cambiar el password de root en mariadb (alumno1234) y dejamos el reto por defecto (excepto volver a cambiar otra vez password de root):

```
sudo mysql_secure_installation
```

Vamos a entrar en mysql y crearemos un usuario y una BBDD para nuestro proyecto:

```
sudo mariadb -u root -p
```

```
CREATE DATABASE tallerlaravel;  
CREATE USER alumno@localhost IDENTIFIED BY 'alumno1234';  
GRANT ALL PRIVILEGES ON tallerlaravel.* TO alumno@localhost;  
FLUSH PRIVILEGES;  
quit;
```

Cambiamos a la carpeta Descargas e instalamos el gestor de dependencia “composer” con:

```
cd ./Descargas  
curl -sS https://getcomposer.org/installer | php  
sudo mv composer.phar /usr/local/bin/composer  
sudo chmod +x /usr/local/bin/composer  
composer --version
```

Ahora cambiamos a la carpeta en que irá nuestra app e instalamos en ella Laravel:

```
cd /var/www/html  
sudo composer create-project --prefer-dist laravel/laravel taller-laravel.local
```

PREPARANDO EL ENTORNO DE DESARROLLO

Ahora vamos a cambiar los permisos de las carpetas de nuestra app para que sea accesible por apache2:

```
sudo chown -R www-data:www-data /var/www/html/taller-laravel.local
sudo chmod -R 775 /var/www/html/taller-laravel.local
```

Y creamos el host virtual en apache2:

```
sudo nano /etc/apache2/sites-available/taller-laravel.local.conf
```

```
<VirtualHost *:80>
    ServerName taller-laravel.local
    ServerAlias www.taller-laravel.local
    DocumentRoot /var/www/html/taller-laravel.local/public

    <Directory /var/www/html/taller-laravel.local/public>
        Options Indexes FollowSymLinks
        AllowOverride All
        Require all granted
    </Directory>

    ErrorLog ${APACHE_LOG_DIR}/error.log
    CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

```
sudo nano /etc/hosts
```

```
127.0.0.1 taller-laravel.local
```

```
sudo a2enmod rewrite
sudo a2ensite taller-laravel.local.conf
sudo systemctl restart apache2
sudo systemctl status apache2
```

PREPARANDO EL ENTORNO DE DESARROLLO

Ahora vamos a cambiar configuraciones en nuestro nuevo proyecto laravel:

```
sudo nano ./taller-laravel.local/.env

APP_NAME=Taller-Laravel
...
APP_TIMEZONE=Europe/Madrid
APP_URL=http://taller-laravel.local

APP_LOCALE=es
...
#DB_CONNECTION=SQLite
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=tallerlaravel
DB_USERNAME=alumno
DB_PASSWORD=alumno1234
```

Y Ahora sí que tenemos la aplicación preparada para una primera ejecución. Pero antes tenemos que ejecutar las **migraciones** iniciales:

```
cd ./taller-laravel.local/
php artisan migrate
```

Para comprobar que ya tenemos nuestro proyecto funcionando, abriremos el navegador desde dentro de Lubuntu y accederemos a la url: <http://taller-laravel.local/>

INSTALAMOS BOOTSTRAP EN LARAVEL

Lo primero que vamos a necesitar es instalar el gestor de dependencias npm:

```
sudo apt install npm
sudo chown -R $USER:www-data /var/www/html/taller-laravel.local
npm install -D bootstrap@5.3.3
npm install -D @popperjs/core
npm install -D sass
npm install -D bootstrap-icons
composer require laravel/ui
php artisan ui bootstrap
npm install bootstrap-icons --save-dev
```

Deberemos modificar el contenido del archivo vite.config.js, para que la ruta que antes apuntaba a css/app.css apunte a a sass/app.scss, debería quedar así:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
  plugins: [
    laravel({
      input: ['resources/sass/app.scss', 'resources/js/app.js'],
      refresh: true,
    }),
  ],
});
```

INSTALAMOS BOOTSTRAP EN LARAVEL

Dentro del archivo resources/js/app.js deberíamos tener esto:

```
// Default Laravel bootstrapper, installs axios
import './bootstrap';
// Added: Actual Bootstrap JavaScript dependency
import 'bootstrap';
// Added: Popper.js dependency for popover support in Bootstrap
import '@popperjs/core';
```

Dentro del archivo resources/sass/app.scss deberías tener esto:

```
@// Fonts
@import url('https://fonts.bunny.net/css?family=Nunito');
// Variables
@import 'variables';
// Bootstrap
@import 'bootstrap/scss/bootstrap';
// Added: Bootstrap icons
@import 'bootstrap-icons/font/bootstrap-icons.css';
```

En el archivo /app/resources/views/welcome.blade.php sustituimos el style donde está todo el estilo de Tailwind CSS por esto:

```
@vite(['resources/sass/app.scss', 'resources/js/app.js'])
```

Ejecutamos:

```
npm install
npm run build
```

INSTALAMOS BOOTSTRAP EN LARAVEL

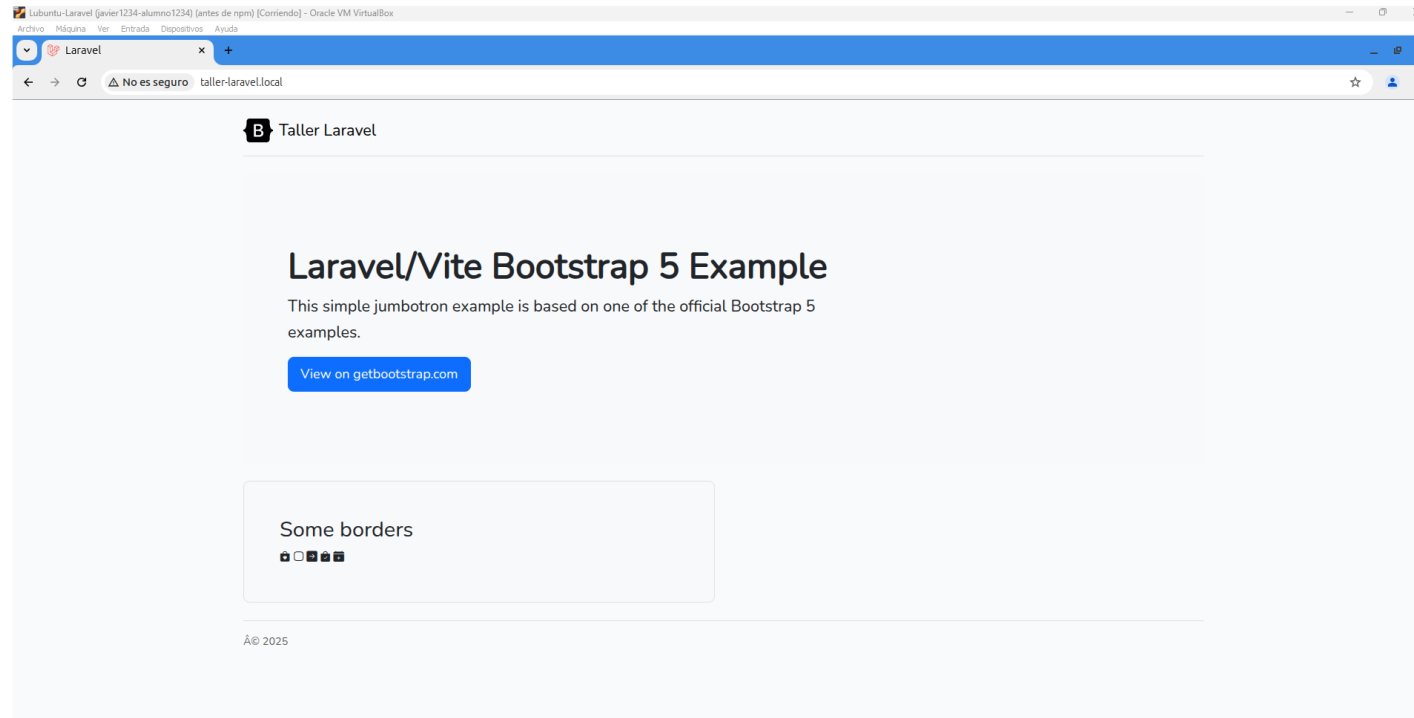
Saldrán muchos warnings pero al final termina de hacer el build. Ahora damos permisos con:

```
sudo chown -R $USER:www-data storage
sudo chown -R $USER:www-data bootstrap/cache
chmod -R 775 storage
chmod -R 775 bootstrap/cache
```

Ahora reemplazamos todo el contenido del body de la vista welcome por: (cod001.txt)

Y probamos desde un navegador a mostrar la url: <http://taller-laravel.local/>

Deberemos ver:



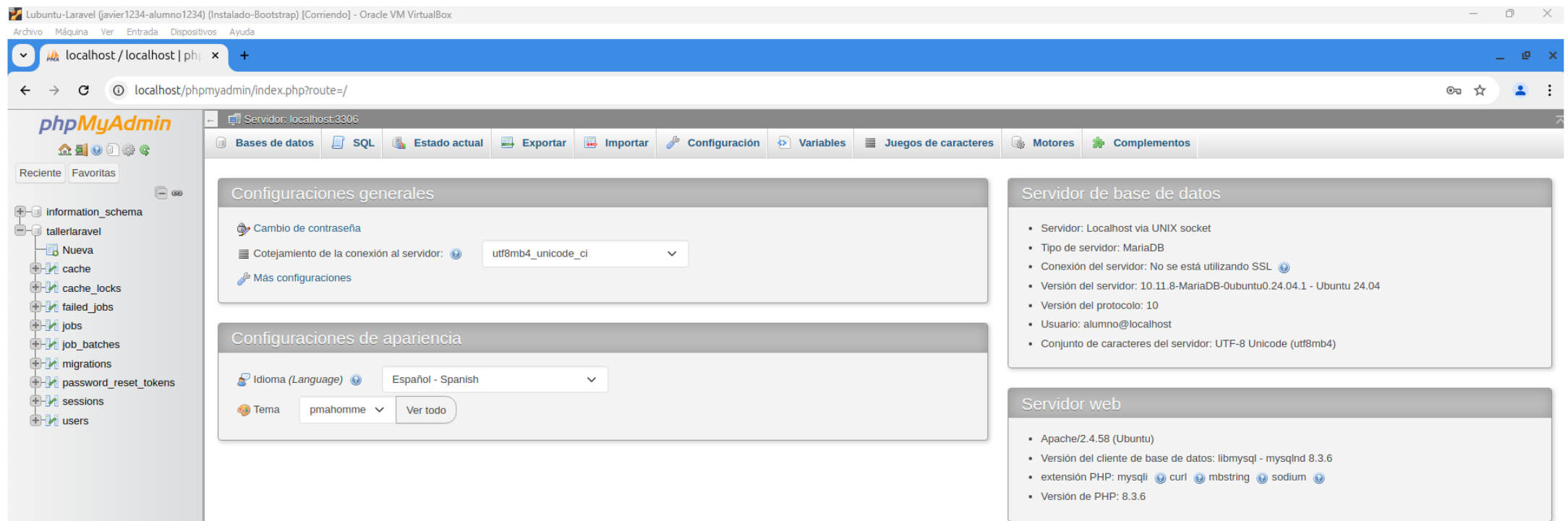
INSTALAMOS PHPMYADMIN

Desde la terminal ejecutaremos:

```
sudo apt-get install phpmyadmin
```

Seleccionamos apache2 como servidor web, le decimos que sí que queremos configurar la BBDD dbconfig-common y le proporcionamos como contraseña: alumno1234

Una vez termina de instalarse, podemos acceder a través de la url: <http://localhost/phpmyadmin/>
Usuario: alumno y contraseña: alumno1234



Configuración

Muchos valores de configuración importantes se definen utilizando el archivo `.env` que existe en la raíz de tu aplicación (revisar archivo).

Tu archivo `.env` no debería ser incluido en el control de versiones de tu aplicación por seguridad.

Podemos acceder a las entradas del fichero `.env` mediante:

```
env('nombre_entrada')
```

(Ejercicio 1)

También puedes acceder a los valores del fichero `config.php` mediante:

```
use Illuminate\Support\Facades\Config;
$value = Config::get('app.timezone');
$value = config('app.timezone');
// Retrieve a default value if the configuration value does not exist...
$value = config('app.timezone', 'Asia/Seoul');
```

(Ejercicio 2)

Estructura de Directorios

El directorio app

La carpeta app, contiene el núcleo del código de la aplicación. Casi todas las clases de una aplicación estarán dentro de esta carpeta.

El directorio config

El directorio config, contiene todos los archivos de configuración de la aplicación. Es una buena idea revisar el contenido de todos estos archivos para familiarizarse con las opciones disponibles.

El directorio database

La carpeta database contiene las migraciones de la base de datos y semillas. Si se desea, se puede utilizar esta carpeta para mantener una base de datos SQLite.

El directorio public

El directorio public contiene el archivo index.php, el cual es el punto de acceso para todas las peticiones que accedan a la aplicación y configura la carga automática. Este directorio contendrá también los recursos JavaScript, CSS, imágenes y otros.

El directorio resources

El directorio resources contiene todas las vistas, así como todos los recursos LESS, SASS, JavaScript y otros archivos sin compilar. Este directorio también incluye los archivos de idioma.

El directorio routes

El directorio routes contiene todas las definiciones de rutas para tu aplicación. Por defecto, se incluyen dos archivos de rutas con Laravel: web.php y console.php

Estructura de Directorios

El directorio storage

El directorio storage contiene todas plantillas Blade compiladas, sesiones y caches basados en archivos y otros archivos generados por el framework. En control de versiones GIT/SVN no es hay que actualizar esta carpeta (estaría en .gitignore).

El directorio vendor

El directorio vendor contiene las dependencias de Composer. Sería el equivalente a node_modules que conocemos en Angular v19. Se puede reconstruir con el comando composer update. En control de versiones GIT/SVN no es hay que actualizar esta carpeta (estaría en .gitignore).

El directorio app/Http

El directorio Http contiene los controladores, middleware y form requests. Casi toda la lógica para gestionar peticiones se encuentra en este directorio.

El directorio app/Models

El directorio Models contiene todas tus clases de modelo Eloquent. Cada tabla de base de datos tiene un "Modelo" correspondiente que se utiliza para interactuar con esa tabla

Se recomienda ampliar esta información con la documentación oficial de Laravel en:

<https://documentacionlaravel.com/docs/11.x/structure>

Instalación Kit de inicio

Antes de proceder a la instalación del Kit de inicio definimos Blade que es un lenguaje de plantillas ligero que laravel renderiza a PHP puro. Mediante Blade podemos utilizar expresiones como:

```
<div>
    @foreach ($users as $user)
        Hello, {{ $user->name }} <br />
    @endforeach
</div>
```

Los Kits de inicio configuran automáticamente tu aplicación con las rutas, controladores y vistas que necesitas para registrar y autenticar a los usuarios de tu aplicación.

Si usáramos Tailwind CSS (predefinido en Laravel v11), usaríamos Laravel Breeze como kit de inicio en nuestro taller, pero como hemos instalado Bootstrap v5, usaremos un kit acorde con Bootstrap.

Desde la terminal ejecutaremos:

```
php artisan ui vue --auth
npm install --force
npm run dev
```

El comando `php artisan ui vue --auth` ha creado todas las vistas necesarias para la autenticación y las ha ubicado en el directorio `resources/views/auth`. Además, ha creado el directorio `resources/views/layouts` que contiene un layout para la aplicación. Todas estas vistas utilizan Bootstrap CSS.

Ahora que se tienen las rutas y vistas configuradas con los controladores de autenticación, se está listo para registrar y autenticar usuarios en la aplicación (<http://taller-laravel.local/register>). **(Ejercicio 3)**

Instalación Kit de inicio

Se puede acceder a nuestra aplicación con seguridad y autenticación. El framework nos lo ha creado todo.

Para que nuestra aplicación sea capaz de enviar los emails recordatorios de contraseñas, es necesario modificar el fichero .env con los datos del servidor de email que vayamos a usar. Por ejemplo:

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.gmail.com
MAIL_PORT=587
MAIL_USERNAME=apps.fjtarrega@gmail.com
MAIL_PASSWORD=XXXXXXXXXXXXXXXXXX
MAIL_ENCRYPTION=tls
```

También podemos usar la herramienta mailpit y para ello configuramos el archivo .env:

```
MAIL_MAILER=smtp
MAIL_HOST=localhost
MAIL_PORT=1025
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
MAIL_FROM_ADDRESS="hello@example.com"
MAIL_FROM_NAME="${APP_NAME}"
```

Con el navegador vamos a la url: <https://github.com/axllent/mailpit/releases/tag/v1.22.2>

Descargamos mailpit-linux-amd64.tar.gz, lo descomprimos en Descargas y ejecutamos:

```
sudo ./mailpit
```

Ahora todos los emails que envíe nuestra aplicación, los captura mailpit y podemos comprobar lo que se envía como si lo hiciera en realidad. En producción cambiaremos la configuración en .env. **(Ejercicio 4)**

LAS RUTAS EN LARAVEL

Las rutas son importantes porque van a ser el mecanismo que tendremos en nuestra aplicación de atender peticiones http. Ante una petición el sistema de enrutado sabrá a dónde dirigir la petición para que se pueda dar una respuesta a la misma.

Ejemplos de rutas:

```
Route::get('/hola', function () {return 'Hola Mundo'}});  
Route::get('/', function () {return view('welcome')}});  
Route::get('/user', [UserController::class, 'index']);  
Route::get('/users', function (Request $request) { dd($request); });  
Route::redirect('/here', '/there');  
Route::view('/welcome', 'welcome');  
Route::view('/welcome', 'welcome', ['nombre' => 'Javier']);  
Route::get('/user/{id}', function (string $id) { return 'User '.$id; });  
Route::get('/posts/{post}/comments/{comment}', function (string $postId, string $commentId) { ... });  
Route::get('/user/{id}', function (Request $request, string $id) { return 'User '.$id; });  
Route::middleware(['first', 'second'])->group(function () {  
    Route::get('/', function () { // Uses first & second middleware... });  
    Route::get('/user/profile', function () { // Uses first & second middleware... });  
});
```

(Ejercicios 5, 6 y 7)

LAS RUTAS EN LARAVEL

El comando Artisan route:list puede proporcionar fácilmente una visión general de todas las rutas que están definidas por tu aplicación. Ejecuta el siguiente comando y comenta lo que aparece:

```
php artisan route:list
```

Si un grupo de rutas utiliza el mismo controlador, puedes usar el método controller para definir el controlador común para todas las rutas dentro del grupo

```
Route::controller(OrderController::class)->group(function () {  
    Route::get('/orders/{id}', 'show');  
    Route::post('/orders', 'store');  
});
```

Laravel resuelve automáticamente los modelos Eloquent definidos en las rutas:

```
Route::get('/users/{user}', function (User $user) { return $user->email; });
```

Laravel inyectará automáticamente la instancia del modelo que tiene un ID que coincide con el valor correspondiente de la URI de la solicitud. Si no se encuentra una instancia de modelo coincidente en la base de datos, se generará automáticamente una respuesta HTTP 404. Por ejemplo, una solicitud a users/1 inyectará la instancia de User de la base de datos que tiene un ID de 1

LOS CONTROLADORES

Los controladores pueden agrupar la lógica de manejo de solicitudes (o peticiones http) relacionadas en una sola clase. Por ejemplo, una clase UserController podría manejar todas las solicitudes entrantes relacionadas con los usuarios, incluyendo mostrar, crear, actualizar y eliminar usuarios. Por defecto, los controladores se almacenan en el directorio app/Http/Controllers. Para crear un controlador, ejecutemos el siguiente comando:

```
php artisan make:controller UserController
```

Vamos a crear en el controlador un método 'show' que recibirá como parámetro de entrada un id y devolverá una vista con los datos del usuario con dicho id:

```
public function show(string $id): View
{
    return view('home', [
        'user' => User::findOrFail($id)
    ]);
}
```

Una vez hecho esto, puedes definir una ruta al método del controlador de la siguiente manera:

```
Route::get('/user/{id}', [App\Http\Controllers\UserController::class, 'show']);
```

Cuando una solicitud entrante coincide con la URI de ruta especificada, se invocará el método show en la clase App\Http\Controllers\UserController y los parámetros de la ruta se pasarán al método. **(Ejercicio 8)**

Controladores de Recursos

Es típico realizar los mismos conjuntos de acciones contra cada recurso en tu aplicación. Por ejemplo, imagina que tu aplicación contiene un modelo Marca y un modelo Ubicacion. Es probable que los usuarios puedan crear, leer, actualizar o eliminar estos recursos. Debido a este caso de uso común, el enrutamiento de recursos de Laravel asigna las rutas típicas de crear, leer, actualizar y eliminar ("CRUD") a un controlador con una sola línea de código. Para crearlo, por ejemplo:

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

A continuación, puedes registrar una ruta de recurso que apunte al controlador::

```
Route::resource('photos', PhotoController::class);
```

Esta única declaración de ruta crea múltiples rutas :

Verbo	URI	Acción	Nombre de Ruta
GET	/photos	index	photos.index
GET	/photos/create	create	photos.create
POST	/photos	store	photos.store
GET	/photos/{photo}	show	photos.show
GET	/photos/{photo}/edit	edit	photos.edit
PUT/PATCH	/photos/{photo}	update	photos.update
DELETE	/photos/{photo}	destroy	photos.destroy

Controladores de Recursos

Puedes registrar múltiples rutas de controladores de recursos a la vez pasando un array al método `resources`, añadamos estas rutas en nuestro `web.php`:

```
Route::resources([
    'marcas' => MarcaController::class,
    'ubicacions' => UbicacionController::class,
]);
```

Puedes usar la opción `--model` al generar el controlador, Esto genera a la vez del controlador, el modelo y añade los parámetros adecuados en los métodos del controlador. Ejecutemos:

```
php artisan make:controller UbicacionController --model=Ubicacion --resource
php artisan make:controller MarcaController --model=Marca --resource
```

Podemos ver las rutas que ya tenemos definidas con:

```
php artisan route:list
```

Si nos fijamos en las rutas de recursos creadas para marcas:

```
GET|HEAD   marcas ..... marcas.index > MarcaController@index
POST       marcas ..... marcas.store > MarcaController@store
GET|HEAD   marcas/create ..... marcas.create > MarcaController@create
GET|HEAD   marcas/{marca} ..... marcas.show > MarcaController@show
PUT|PATCH marcas/{marca} ..... marcas.update > MarcaController@update
DELETE     marcas/{marca} ..... marcas.destroy > MarcaController@destroy
GET|HEAD   marcas/{marca}/edit ..... marcas.edit > MarcaController@edit
```

PETICIONES Y RESPUESTAS HTTP

Vemos que cuando se reciba una petición GET en la url `http://taller-laravel.local/marcas` se ejecutará el método `index` de la clase `MarcaController`.

Lo normal en ese método `index` sería que el controlador llamara al modelo para recuperar todas las marcas de la BBDD y con esa información diera una respuesta. Esta respuesta debería ser una vista que nos muestre por pantalla la relación de marcas y además tuviera un botón que permitiera crear una nueva marca, y para cada marca dos botones para poder modificar y borrar la marca.

Al pulsar el botón “Nueva Marca” se generaría una petición GET en `marcas/creáre` y se ejecutará el método `create` de la clase `MarcaController`.

Lo normal en ese método sería comprobar si el usuario tiene permisos y en caso afirmativo, devolver por pantalla una vista que contendría un formulario con un botón de aceptar y cancelar.

Si se pulsa el botón cancelar se haría un GET a `/marcas` que volvería a mostrar el listado de marcas.

Si se pulsa el botón aceptar se ejecuta un submit del formulario que genera un POST en la url `/marcas` que ejecutará el método `store` de la clase `MarcaController`.

Lo normal en ese método `create` sería que el controlador recibiera como parámetro la marca a crear y llamara al modelo para que la inserte en la BBDD y diera una respuesta. Esta respuesta debería ser la vista que nos muestre por pantalla la relación de marcas en la que ya aparecerá la nueva marca creada.

PETICIONES Y RESPUESTAS HTTP

Al pulsar el botón “Modificar Marca” se generaría una petición GET en `marcas/{marca}/edit` y se ejecutará el método `edit` de la clase `MarcaController`.

Lo normal en ese método sería comprobar si el usuario tiene permisos y en caso afirmativo, acceder al modelo para recuperar los datos de la marca que ha recibido como parámetro y devolver por pantalla una vista que contendría un formulario modificable con los datos recibidos del modelo, con dos botones de guardar y cancelar.

Si se pulsa el botón cancelar se haría un GET a `/marcas` que volvería a mostrar el listado de marcas.

Si se pulsa el botón guardar se ejecuta un submit del formulario que genera un PUT en la url `/marcas/{marca}` que ejecutará el método `update` de la clase `MarcaController`.

Lo normal en ese método `update` sería que el controlador recibiera como parámetro la marca a modificar y llamara al modelo para que la modifique en la BBDD y diera una respuesta. Esta respuesta debería ser la vista que nos muestre por pantalla la relación de marcas en la que ya aparecerá la marca modificada.

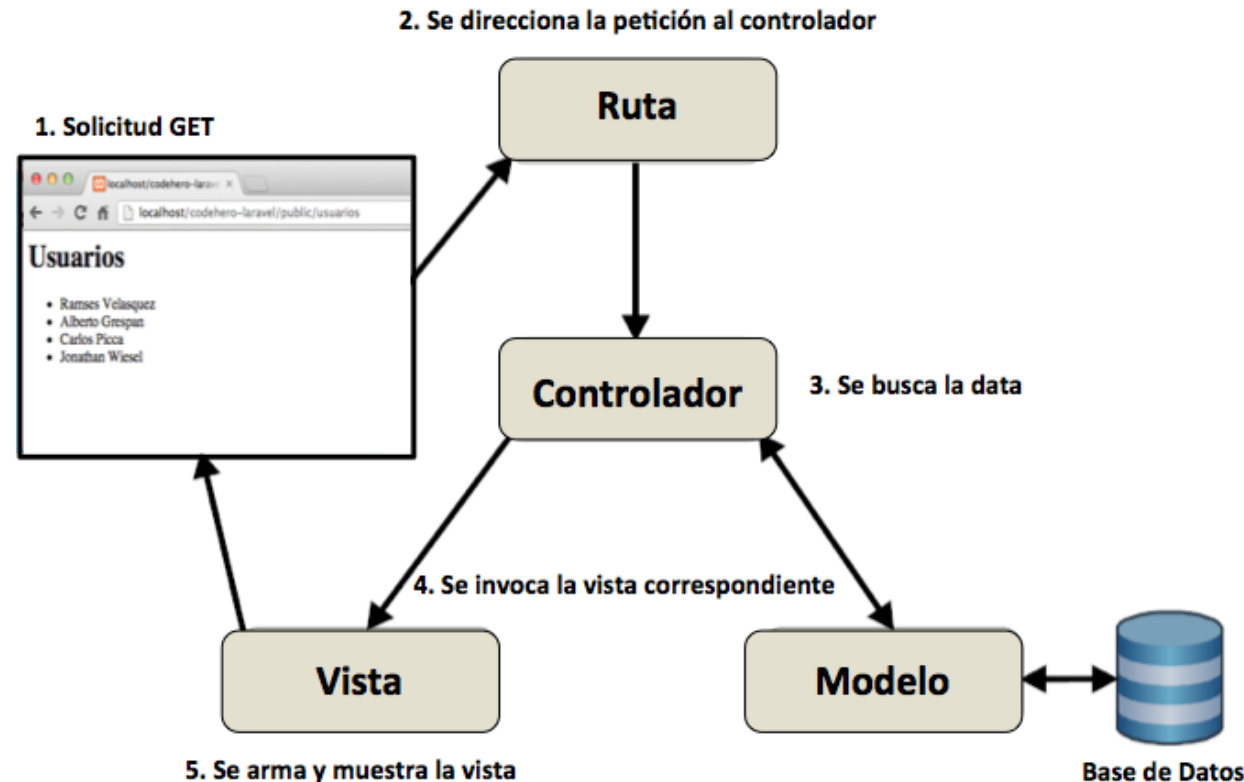
Al pulsar el botón “Borrar Marca” se generaría una petición GET en `marcas/{marca}` y se ejecutará el método `show` de la clase `MarcaController`.

Lo normal en ese método sería comprobar si el usuario tiene permisos y en caso afirmativo, acceder al modelo para recuperar los datos de la marca que ha recibido como parámetro y devolver por pantalla una vista que contendría un formulario de sólo lectura con los datos recibidos del modelo y con dos botones de borrar y cancelar.

PETICIONES Y RESPUESTAS HTTP

Si se pulsa el botón cancelar se haría un GET a /marcas que volvería a mostrar el listado de marcas.
Si se pulsa el botón borrar se ejecuta un submit del formulario que genera un DELETE en la url /marcas/{marca} que ejecutará el método destroy de la clase MarcaController.

Lo normal en ese método destroy sería que el controlador recibiera como parámetro la marca a eliminar y llamara al modelo para que la borre en la BBDD y diera una respuesta. Esta respuesta debería ser la vista que nos muestre por pantalla la relación de marcas en la que ya no aparecerá la marca eliminada.



LAS VISTAS

Las vistas separan la lógica de tu controlador / aplicación de tu lógica de presentación y se almacenan en el directorio `resources/views`. Al usar Laravel, las plantillas de vista suelen escribirse utilizando el lenguaje de plantillas Blade.

Puedes crear una vista colocando un archivo con la extensión `.blade.php` en el directorio `resources/views` de tu aplicación o utilizando el comando Artisan:

```
php artisan make:view greeting
```

Las plantillas Blade contienen HTML así como directivas Blade que te permiten fácilmente devolver valores, crear sentencias "if", iterar sobre datos y más.

Las vistas también pueden estar anidadas dentro de subdirectorios del directorio `resources/views`. Se puede usar notación "punto" para hacer referencia a las vistas anidadas. Por ejemplo, si tu vista se almacena en `resources/views/marca/lista.blade.php`, puedes devolverla desde una de las rutas / controladores de tu aplicación de la siguiente manera:

```
return view('marca.lista', $data);
```

Vamos a crear una vista para listar las marcas en nuestra aplicación con el siguiente comando:

```
php artisan make:view marca/lista
```

LAS VISTAS

Puedes pasar un array de datos a las vistas para hacer que esos datos estén disponibles en la vista:

```
return view('greetings', ['name' => 'Victoria']);
```

Al pasar información de esta manera, los datos deben ser un array con pares clave / valor. Después de proporcionar datos a una vista, puedes acceder a cada valor dentro de tu vista utilizando las claves de los datos, como:

```
<?php echo $name; ?>
```

Vamos a modificar el método index de la clase MarcaController:

```
public function index()
{
    //TODO Almacenar en $marcas todos los datos de las marcas de la BBDD
    $marcas = array(['id' => 1, 'marca' => 'Marca1'], ['id' => 2, 'marca' => 'Marca2'], ['id' => 3, 'marca'
=> 'Marca3']);
    return view('marca.lista', ['marcas' => collect($marcas)]);
}
```

Y modificamos también marca/lista.blade.php:

```
<div><p>Recibido del controlador la coleccion de marcas: {{ $marcas[0]['id'] }} {{ $marcas[0]['marca'] }}
</p></div>
```

Y si accedemos a <http://taller-laravel.local/marcas> vemos vista con los datos del controlador.

Blade

Sentencias If

Puedes construir declaraciones if utilizando las directivas @if, @elseif, @else y @endif. Estas directivas funcionan de manera idéntica a sus equivalentes en PHP:

```
@if (count($records) === 1)
    I have one record!
}elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

Podemos ver cómo se usa esta directiva en `resources/views/layouts/app.blade.php` (línea 40)

```
@empty($records)
    // $records is "empty"...
@endempty
```

Directivas de Autenticación

Las directivas @auth y @guest se pueden usar para determinar rápidamente si el usuario actual está autenticado o es un invitado:

Blade

```
@auth
    // The user is authenticated...
@endauth

@guest
    // The user is not authenticated...
@endguest
```

Podemos ver cómo se usa esta directiva en `resources/views/layouts/app.blade.php` (línea 39)

```
@empty($records)
    // $records is "empty"...
@endempty
```

Directivas de Sección

Puedes determinar si una sección de herencia de plantilla tiene contenido utilizando la directiva `@hasSection`:

```
@hasSection('navigation')
    <div class="pull-right">
        @yield('navigation')
    </div>

    <div class="clearfix"></div>
@endif
```

Blade

Directivas de Sesión

@session se puede usar para determinar si existe un valor de sesión:

```
@session('status')
    <div class="p-4 bg-green-100">
        {{ $value }}
    </div>
@endsession
```

Podemos ver cómo se usa esta directiva en `resources/views/home.blade.php` línea 11

Sentencias Switch

Las declarativas de Switch se pueden construir utilizando las directivas @switch, @case, @break, @default y @endswitch:

```
@switch($i)
    @case(1)
        First case...
    @break

    @case(2)
        Second case...
    @break

    @default
        Default case...
@endswitch
```

Blade

Bucles

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>No users</p>
@endforelse

@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

También finalizar el bucle utilizando las directivas @continue y @break:

```
@foreach ($users as $user)
    @if ($user->type == 1)
        @continue
    @endif

    <li>{{ $user->name }}</li>

    @if ($user->number == 5)
        @break
    @endif
@endforeach
```

Blade

Vamos a utilizar estas directivas en nuestra vista `resources/views/marca/lista.blade.php` de la siguiente forma:

```
@if ($marcas->isEmpty())
    <div>
        <h3>No hay Marcas</h3>
    </div>
@else
    <table>
        <tr>
            <th>ID</th>
            <th>MARCA</th>
        </tr>
        @foreach ($marcas as $marca)
            <tr>
                <td>{{ $marca['id'] }}</td>
                <td>{{ $marca['marca'] }}</td>
            </tr>
        @endforeach
    </table>
@endif
```

Incluyendo Subvistas

La directiva `@include` de Blade te permite incluir una vista de Blade desde otra vista. Todas las variables que están disponibles para la vista padre estarán disponibles para la vista incluida:

Blade

```
<div>
    @include('shared.errors')

    <form>
        <!-- Form Contents -->
    </form>
</div>
```

Aunque la vista incluida heredará todos los datos disponibles en la vista padre, también puedes pasar un array de datos adicionales que deberían estar disponibles para la vista incluida:

```
@include('view.name', ['status' => 'complete'])
```

PHP en Crudo

En algunas situaciones, es útil incrustar código PHP en tus vistas. Puedes usar la directiva @php de Blade para ejecutar un bloque de PHP puro dentro de tu plantilla:

```
@php
    $counter = 1;
@endphp
```

O, si solo necesitas usar PHP para importar una clase, puedes usar la directiva @use:

```
@use('App\Models\Flight')
```

Blade - Construyendo Layouts

Veamos esto con un ejemplo simple. Dado que la mayoría de las aplicaciones web mantienen el mismo diseño general en varias páginas, es conveniente definir este diseño como una sola vista de Blade:

```
<!-- resources/views/layouts/app.blade.php -->
<html>
  <head>
    <title>App Name - @yield('title')</title>
  </head>
  <body>
    <div class="container">
      @yield('content')
    </div>
  </body>
</html>
```

Extendiendo un Layout

Al definir una vista hija, utiliza la directiva @extends de Blade para especificar qué diseño debería "heredar" la vista hija:

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

@section('title', 'Page Title')

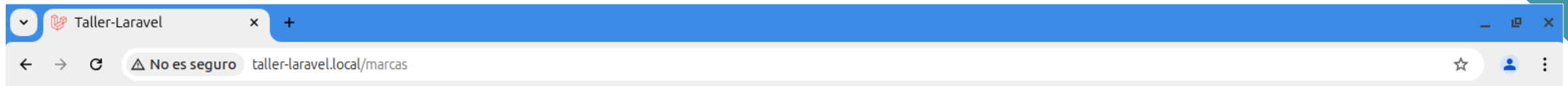
@section('content')
  <p>This is my body content.</p>
@endsection
```

Blade - Construyendo Layouts

Vamos a utilizar estas @extends en nuestra vista resources/views/marca/lista.blade.php y veremos cómo cambia completamente el aspecto de la página:

(cod002.txt)

```
@extends('layouts.app')/@section('content')/@if/@else/@foreach ($marcas as $marca)/@endforeach/@endif/@endsection
```



Taller-Laravel

Javier ▾

Relación de Marcas

Nueva Marca

Volver

Acciones	ID	MARCA
Modif. Borrar	1	Marca1
Modif. Borrar	2	Marca2
Modif. Borrar	3	Marca3

(Ejercicio 9)

Blade - Construyendo Layouts

Generando URL

El helper url se puede utilizar para generar URL para tu aplicación. La URL generada utilizará automáticamente el esquema (HTTP o HTTPS) y el host definido en la configuración de la aplicación:

```
$post = 1;
echo url("/posts/{$post}");
// http://taller-laravel.local/posts/1
```

Accediendo a la URL Actual

```
// Get the current URL without the query string...
echo url()->current();

// Get the current URL including the query string...
echo url()->full();

// Get the full URL for the previous request...
echo url()->previous();
```

Con lo que ya conocemos de plantillas Blade y de generación de rutas y URLs, vamos a añadir a nuestra aplicación un menú de opciones de Bootstrap para que acceda a la relación de marcas y vamos a dar funcionalidad a los botones que podamos del listado de marcas:

CRUD – Marcas (sin BBDD)

1. Eliminar la vista welcome.blade.php
2. Cambiar las rutas en web.php de la siguiente forma:

```
<?php
```

```
use Illuminate\Support\Facades\Route;  
use Illuminate\Support\Facades\Auth;  
use App\Http\Controllers\MarcaController;  
use App\Http\Controllers\UbicacionController;
```

```
Auth::routes();  
Route::middleware(['auth'])->group(function () {  
    Route::redirect('/', '/home');  
    Route::get('/home', [App\Http\Controllers\HomeController::class, 'index'])->name('home');  
    Route::resources([  
        'marcas' => MarcaController::class,  
        'ubicacions' => UbicacionController::class,  
    ]);  
});
```

3. En resources/views/home.blade.php cambiamos:

```
<div class="card">  
    <div class="card-header">{{ __('INICIO') }}</div>  
  
    <div class="card-body">  
        @if (session('status'))  
            <div class="alert alert-success" role="alert">  
                {{ session('status') }}  
            </div>  
        @endif  
        {{ __('Bienvenido ') }}de nuevo {{ Auth::user()->name }}  
    </div>  
</div>
```

CRUD – Marcas (sin BBDD)

4. En resources/views/layouts/app.blade.php añadimos la carga de Fontawesome::

```
<!-- Fonts -->
<link rel="dns-prefetch" href="//fonts.bunny.net">
<link href="https://fonts.bunny.net/css?family=Nunito" rel="stylesheet">
<link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.8.1/css/all.css" integrity="sha384-
50oBUHEmvpQ+1lW4y57PTFmhCaXp0ML5d60M1M7uH2+nqUivzIebhnd0JK28anvf" crossorigin="anonymous">
<!-- Scripts -->
```

Y añadimos el menú de opciones Archivos:

```
<!-- Left Side Of Navbar -->
<ul class="navbar-nav me-auto">
    @guest
    @else
        <li class="nav-item dropdown">
            <a class="nav-link dropdown-toggle" href="#" role="button" data-bs-toggle="dropdown" aria-
            expanded="false">Archivos</a>
            <ul class="dropdown-menu">
                <li><a class="dropdown-item" href="{{ url('/marcas') }}">Marcas</a></li>
                <li><a class="dropdown-item" href="{{ url('/ubicaciones') }}">Ubicaciones</a></li>
            </ul>
        </li>
    @endguest
</ul>
```

CRUD – Marcas (sin BBDD)

5. En resources/views/marca/lista.blade.php cambiamos:

```
<a href="{{ route('marcas.create') }}" class="btn btn-primary text-white"><i class="fa fa-plus"></i>Nueva  
Marca</a>  
<a href="{{ route('home') }}" class="btn btn-primary text-white ms-1"><i class="fa fa-arrow-  
left"></i>Volver</a>
```

Y

```
<a href="marcas/{{ $marca['id'] }}/edit" class="btn btn-sm btn-warning me-1"><i class="fa fa-edit"></i>  
Modif.</a>  
<a href="{{ route('marcas.destroy', $marca['id']) }}" class="btn btn-sm btn-danger"><i class="fa fa-  
times"></i> Borrar</a>
```

6. En app/Http/Controllers/MarcaController.php cambiamos:

```
public function create()  
{  
    return View('marca.create');  
}
```

7. Los botones de modificar y borrar marcas no pueden ser funcionales hasta que no tengamos la tabla de marcas con datos en la BBDD

(Ejercicio 10)

Blade - Formularios en las Plantillas

Campo CSRF

Siempre que definas un formulario HTML en tu aplicación, debes incluir un campo de token CSRF oculto en el formulario para que el middleware de protección CSRF pueda validar la solicitud. Puedes usar la directiva @csrf de Blade para generar el campo de token:

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

Campo de Método

Dado que los formularios HTML no pueden hacer solicitudes PUT, PATCH o DELETE, necesitarás añadir un campo oculto _method para simular estos verbos HTTP. La directiva @method de Blade puede crear este campo por ti:

```
<form action="/foo/bar" method="POST">  
    @method('PUT')  
    ...  
</form>
```

Con lo que hemos visto de los formularios de plantillas Blade y de generación de rutas y URLs, vamos a crear el formulario para Nueva Marca:

Formulario para Nueva Marca

1. Creamos la vista resources/views/marca/create.blade.php con el formulario para el alta

```
@extends('layouts.app')
@section('content')
```

```
<div class="container">
  <div class="row justify-content-center mt-3">
    <div class="card col-lg-8">
      <div class="card-header d-inline-flex justify-content-between">
        <h2>Nueva Marca</h2>
        <div class="navbar-text">
          <a href="{{route('marcas.index')}}" class="btn btn-primary text-white"><i class="fa fa-
arrow-left"></i> Volver</a>
        </div>
      </div>
      <div class="card-body">
        <form method="POST" action="{{route('marcas.store')}}" accept-charset="UTF-8">
          @csrf
          @include('marca.fieldsform', ['items'=> 'create'])
          <input name="grabar" id="grabar" class="btn btn-primary btn-sm mt-3" type="submit"
value="Grabar">
        </form>
      </div>
    </div>
  </div>
</div>
@endsection
```

Formulario para Nueva Marca

2. Creamos el archivo resources/views/marca/fieldsform.blade.php con los campos del formulario

```
<div class="form-group">
    <label for="nombre">Marca:</label>
    @if ($items == 'create')
        <input id="nombre" class="form-control" placeholder="Introduce la marca" maxlength="50" name="nombre"
type="text">
    @endif
</div>
```

3. Implementamos el método store de app/Http/Controllers/MarcaController.php:

```
public function store(Request $request)
{
    // TODO guardamos la marca...
    return redirect()->route('marcas.index');
}
```

SESIÓN HTTP - Datos Flash

A veces es posible que desees almacenar elementos en la sesión para la siguiente solicitud. Puedes hacerlo utilizando el método flash. Los datos almacenados en la sesión utilizando este método estarán disponibles de inmediato y durante la solicitud HTTP posterior. Después de la solicitud HTTP posterior, los datos en flash serán eliminados. Los datos en flash son principalmente útiles para mensajes de estado de corta duración:

```
$request->session()->flash('status', 'Task was successful!');
```

Eliminar Datos:

```
$request->session()->flush();
```

Con lo que hemos visto de las sesiones vamos a añadir a nuestro ejemplo una gestión de mensajes utilizando datos flash de la sesión.

Vamos a crear un fichero resources/views/layouts/errores.blade.php que contendrá: **(cod003.txt)**

Y ahora podemos incluir este código en todas las vistas en que podamos querer mostrar un mensaje por ejemplo en resources/views/marca/lista.blade.php:

```
<div class="card-body">  
    @include('layouts.errores')  
    @if ($marcas->isEmpty())
```


SESIÓN HTTP - Datos Flash

Y vamos a enviar un mensaje cuando se cree una nueva marca:

```
public function store(Request $request)
{
    // TODO guardamos la marca...
    Session::flash('success', 'La marca se ha creado satisfactoriamente');
    return redirect()->route('marcas.index');
}
```

Relación de Marcas

[+ Nueva Marca](#)[← Volver](#)

¡Correcto! : La marca se ha creado satisfactoriamente

Acciones	ID	MARCA
✎ Modif. ✖ Borrar	1	Marca1
✎ Modif. ✖ Borrar	2	Marca2
✎ Modif. ✖ Borrar	3	Marca3

De esta forma podemos enviar mensajes entre controladores y vistas utilizando los datos flash de la sesión

(Ejercicio 12)

VALIDACIÓN

Laravel ofrece varias formas diferentes de validar los datos entrantes de tu aplicación. Lo más común es usar el método `validate` disponible en todas las solicitudes HTTP entrantes.

Vamos a utilizar nuestro método `store` con la lógica para validar la nueva marca introducida. Si las reglas de validación pasan, tu código continuará ejecutándose normalmente; sin embargo, si la validación falla, se lanzará una excepción y se enviará automáticamente la respuesta de error adecuada al usuario.

```
$validator = Validator::make($request->all(), ['nombre' => 'required|max:50|unique:marcas',], ['nombre.unique'
=> 'No se ha grabado porque la marca introducida ya la has usado antes. Introduce otra por favor.',
'nombre.required' => 'Introduce la marca por favor.']);
if ($validator->fails()) {
    return redirect('marcas/create') ->withErrors($validator) ->withInput();
}
```

Puedes consultar todas las reglas de validación disponibles en:

<https://documentacionlaravel.com/docs/11.x/validation#available-validation-rules>

Si los campos de la solicitud entrante no pasan las reglas de validación todos los errores de validación y la entrada de la solicitud se flashearán automáticamente a la sesión.

Una variable `$errors` se comparte con todas las vistas de tu aplicación y siempre estará disponible en tus vistas.

En nuestro ejemplo, el usuario será redirigido al método `create` de nuestro controlador cuando la validación falle, que a su vez nos muestra la vista `marca/create.blade.php` en la que vamos a añadir:

VALIDACIÓN

```
@include('layouts.errores')  
<form method="POST" action="{{route('marcas.store')}}" accept-charset="UTF-8">
```

Y con esto ya nos mostraría los errores a través de los datos flas generados en la sesión

Nueva Marca

[← Volver](#)

Errores:

- Introduce la marca por favor.

Marca:

[Grabar](#)

Repoblando Formularios

Cuando Laravel genera una respuesta de redirección debido a un error de validación, el framework automáticamente flashea toda la entrada de la solicitud a la sesión.

Para recuperar la entrada pasada del request anterior, invoca el método old:

```
$title = $request->old('title');
```

Si estás mostrando entrada anterior dentro de una plantilla Blade, es más conveniente usar el helper old para rellenar de nuevo el formulario:

```
<input type="text" name="title" value="{{ old('title') }}">
```

AUTENTICACIÓN

Para saber el usuario que está logeado:

```
use Illuminate\Support\Facades\Auth;

// Retrieve the currently authenticated user...
$user = Auth::user();

// Retrieve the currently authenticated user's ID...
$id = Auth::id();
```

También a través del método user de la solicitud:

```
$user = $request->user();
```

Para determinar si el usuario que realiza la solicitud HTTP entrante está autenticado, puedes usar el método check en la fachada Auth:

```
use Illuminate\Support\Facades\Auth;

if (Auth::check()) {
    // The user is logged in...
}
```

Servicios de Autenticación API de Laravel

Laravel ofrece dos paquetes opcionales para ayudarte a gestionar tokens API y autenticar solicitudes realizadas con tokens API: Passport y Sanctum.

Nosotros más adelante usaremos Sanctum para la API RESTFull que crearemos.

BASE DE DATOS - Migraciones

Laravel puede utilizar SQL en bruto (RAW), un constructor de consultas (Query Builder) y el ORM Eloquent. Actualmente, Laravel ofrece soporte de primera mano para cinco bases de datos:

- ✓ MariaDB 10.3+ (Version Policy)
- ✓ MySQL 5.7+ (Version Policy)
- ✓ PostgreSQL 10.0+ (Version Policy)
- ✓ SQLite 3.26.0+
- ✓ SQL Server 2017+ (Version Policy)

La configuración para los servicios de base de datos de Laravel se encuentra en el archivo de configuración `config/database.php`.

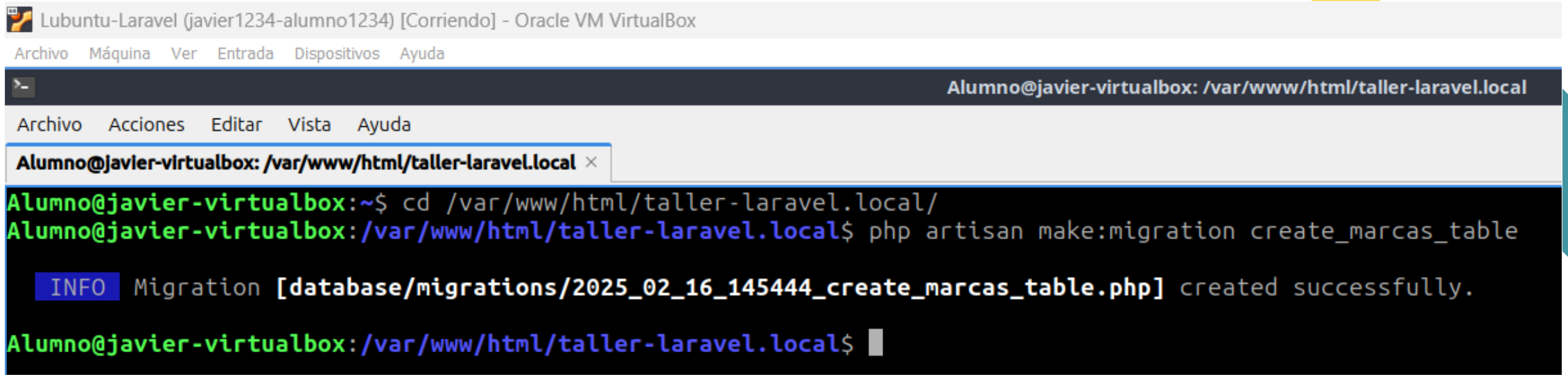
Migraciones

Las migraciones son como control de versiones para tu base de datos, lo que permite a tu equipo definir y compartir la definición del esquema de la base de datos de la aplicación. Permiten crear y modificar tablas y columnas de base de datos.

Vamos a crear una migración para nuestra tabla de marcas:

```
php artisan make:migration create_marcas_table
```

BASE DE DATOS - Migraciones



```
Lubuntu-Laravel (javier1234-alumno1234) [Corriendo] - Oracle VM VirtualBox
Archivo  Máquina  Ver  Entrada  Dispositivos  Ayuda

Alumno@javier-virtualbox: /var/www/html/taller-laravel.local

Archivo  Acciones  Editar  Vista  Ayuda

Alumno@javier-virtualbox: /var/www/html/taller-laravel.local x
Alumno@javier-virtualbox:~$ cd /var/www/html/taller-laravel.local/
Alumno@javier-virtualbox: /var/www/html/taller-laravel.local$ php artisan make:migration create_marcas_table

INFO Migration [database/migrations/2025_02_16_145444_create_marcas_table.php] created successfully.

Alumno@javier-virtualbox: /var/www/html/taller-laravel.local$
```

Si Laravel puede determinar el nombre de la tabla a partir del nombre de la migración, Laravel rellenará automáticamente el archivo de migración generado con la tabla especificada.

Estructura de la Migración

Una clase de migración contiene dos métodos: up y down. El método up se utiliza para agregar nuevas tablas, columnas o índices a tu base de datos, mientras que el método down debe revertir las operaciones realizadas por el método up.

vamos a crear la tabla de marcas con el siguiente código en la migración que acabamos de crear:

BASE DE DATOS - Migraciones

```
public function up(): void
{
    Schema::create('marcas', function (Blueprint $table) {
        $table->id();
        $table->string('nombre',50);
        $table->timestamps();
    });
}
public function down(): void
{
    Schema::dropIfExists('marcas');
}
```

Ejecutando Migraciones

Para ejecutar todas tus migraciones pendientes, ejecuta el comando:

```
php artisan migrate
```

Si deseas ver qué migraciones se han ejecutado hasta ahora, puedes usar el comando:

```
php artisan migrate:status
```

Revirtiendo Migraciones

Para deshacer la última operación de migración:

```
php artisan migrate:rollback
```

Puedes revertir un número limitado de migraciones proporcionando la opción step:

```
php artisan migrate:rollback --step=5
```


BASE DE DATOS - Migraciones

Puedes revertir todas las migraciones de tu aplicación:

```
php artisan migrate:reset
```

Este comando vuelve a crear toda tu base de datos:

```
php artisan migrate:refresh
```

Puedes utilizar la opción step al comando refresh. Por ejemplo, el siguiente comando deshará y volverá a migrar las últimas cinco migraciones:

```
php artisan migrate:refresh --step=5
```

El comando migrate:fresh eliminará todas las tablas de la base de datos y luego ejecutará el comando migrate:

```
php artisan migrate:fresh
```

Crear Tablas en migraciones

Para crear una nueva tabla de base de datos, hemos utilizado el método create:

```
Schema::create('marcas', function (Blueprint $table) {  
    $table->id();  
    $table->string('nombre',50);  
    $table->timestamps();  
});
```

BASE DE DATOS - Migraciones

Puedes revertir todas las migraciones de tu aplicación:

```
php artisan migrate:reset
```

Este comando vuelve a crear toda tu base de datos:

```
php artisan migrate:refresh
```

Puedes utilizar la opción step al comando refresh. Por ejemplo, el siguiente comando deshará y volverá a migrar las últimas cinco migraciones:

```
php artisan migrate:refresh --step=5
```

El comando migrate:fresh eliminará todas las tablas de la base de datos y luego ejecutará el comando migrate:

```
php artisan migrate:fresh
```

Crear Tablas en migraciones

Para crear una nueva tabla de base de datos, hemos utilizado el método create:

```
Schema::create('marcas', function (Blueprint $table) {  
    $table->id();  
    $table->string('nombre',50);  
    $table->timestamps();  
});
```

<https://documentacionlaravel.com/docs/11.x/migrations>

(Ejercicio 14)

BASE DE DATOS - Migraciones

Migración para tabla de Ordenadores (con claves ajenas)

Laravel también proporciona soporte para la creación de restricciones de clave foránea, que se utilizan para forzar la integridad referencial a nivel de base de datos. Por ejemplo, definamos una columna `idmarca` en la tabla `ordenadors` que haga referencia a la columna `id` en la tabla `marcas`; y también, definamos una columna `idubicacion` en la tabla `ordenadors` que haga referencia a la columna `id` en la tabla de `ubicacions`:

```
public function up(): void
{
    Schema::create('ordenadors', function (Blueprint $table) {
        $table->id();
        $table->string('numero', 20)->nullable($value = true);
        $table->unsignedBigInteger('idmarca');
        $table->foreign('idmarca', 'fk_ordenadors_marcas')->references('id')->on('marcas')->onDelete('restrict');
        $table->string('modelo', 20)->nullable($value = true);
        $table->unsignedBigInteger('idubicacion');
        $table->foreign('idubicacion', 'fk_ordenadors_ubicacions')->references('id')->on('ubicacions')->onDelete('restrict');
        $table->string('tppc', 20)->nullable($value = true);
        $table->string('numserie', 25)->nullable($value = true);
        $table->string('red', 20)->nullable($value = true);
        $table->macAddress('maclan')->nullable($value = true);
        $table->ipAddress('iplan')->nullable($value = true);
        $table->macAddress('macwifi')->nullable($value = true);
        $table->ipAddress('ipwifi')->nullable($value = true);
        $table->string('hd1', 50)->nullable($value = true);
        $table->string('hd2', 50)->nullable($value = true);
        $table->longText('observaciones')->nullable($value = true);
        $table->timestamps();
    });
}
```

(Ejercicio 15)

SQL en bruto (RAW)

Para ejecución de instrucciones SQL en bruto o RAW, podemos usar la fachada DB que nos proporciona métodos para cada tipo de consulta: select, update, insert, delete y statement

Para ejecutar una consulta SELECT básica, puedes usar el método select en la facade DB, por ejemplo:

```
use Illuminate\Support\Facades\DB;

$users = DB::select('select * from users');

foreach ($users as $user) {
    echo $user->name;
}
```

Siempre devolverá un array de resultados.

El enlace de parámetros proporciona protección contra inyecciones SQL

```
$users = DB::select('select * from users where active = ?', [1]);
```

A veces, tu consulta a la base de datos puede resultar en un solo valor escalar

```
$burgers = DB::scalar(
    "select count(case when food = 'burger' then 1 end) as burgers from menu"
);
```

En lugar de usar ? para representar tus enlaces de parámetros, puedes ejecutar una consulta utilizando enlaces con nombre:

```
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

SQL en bruto (RAW)

Para ejecutar una declaración insert, puedes usar el método insert :

```
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Marc']);
```

El método update debe utilizarse para actualizar registros existentes en la base de datos. El número de filas afectadas por la instrucción es devuelto:

```
$affected = DB::update('update users set votes = 100 where name = ?', ['Anita']);
```

El método delete debe utilizarse para eliminar registros de la base de datos. Al igual que update, el número de filas afectadas será devuelto

```
$deleted = DB::delete('delete from users');
```

Query Builder

Con el constructor de consultas de base de datos de Laravel no es necesario limpiar o sanear las cadenas pasadas como parámetros.

Recuperando Todas las Filas de una Tabla:

```
$users = DB::table('users')->get();
```

Puedes acceder al valor de cada columna accediendo a la columna como una propiedad del objeto:

```
foreach ($users as $user) {  
    echo $user->name;  
}
```

Recuperando una Sola Fila / Columna de una Tabla

```
$user = DB::table('users')->where('name', 'John')->first();  
$email = $user->email;
```

```
$user = DB::table('users')->where('name', 'John')->firstOrFail();
```

```
$email = DB::table('users')->where('name', 'John')->value('email');
```

```
$user = DB::table('users')->find(3);
```

Query Builder

Si deseas obtener una coleccion que contenga los valores de una sola columna, puedes usar el método pluck.

```
$titles = DB::table('users')->pluck('title');  
foreach ($titles as $title) {  
    echo $title;  
}
```

Para recuperar valores agregados como count, max, min, avg y sum:

```
$users = DB::table('users')->count();  
$price = DB::table('orders')->max('price');
```

Otros:

```
$users = DB::table('users')  
    ->select('name', 'email as user_email')  
    ->get();
```

```
$users = DB::table('users')  
    ->join('contacts', 'users.id', '=', 'contacts.user_id')  
    ->join('orders', 'users.id', '=', 'orders.user_id')  
    ->select('users.*', 'contacts.phone', 'orders.price')  
    ->get();
```

Query Builder

```
$users = DB::table('users')  
    ->where('votes', '=', 100)  
    ->where('age', '>', 35)  
    ->get();
```

```
$users = DB::table('users')  
    ->where('votes', '>', 100)  
    ->orWhere('name', 'John')  
    ->get();
```

```
$users = DB::table('users')  
    ->orderBy('name', 'desc')  
    ->get();
```

```
$users = DB::table('users')  
    ->groupBy('account_id')  
    ->having('account_id', '>', 100)  
    ->get();
```

El generador de consultas también proporciona un método insert que se puede usar para insertar registros en la tabla de la base de datos. El método insert acepta un array de nombres de columnas y valores:

```
DB::table('users')->insert([  
    'email' => 'kayla@example.com',  
    'votes' => 0  
]);
```


Query Builder

El generador de consultas puede actualizar registros existentes utilizando el método `update`. Acepta un array de pares de columna y valor que indican las columnas a actualizar y devuelve el número de filas afectadas:

```
$affected = DB::table('users')
    ->where('id', 1)
    ->update(['votes' => 1]);

DB::table('users')
    ->updateOrInsert(
        ['email' => 'john@example.com', 'name' => 'John'],
        ['votes' => '2']
    );
```

El método `delete` del generador de consultas se puede utilizar para eliminar registros de la tabla. Devuelve el número de filas afectadas:

```
$deleted = DB::table('users')->delete();
$deleted = DB::table('users')->where('votes', '>', 100)->delete();
```

Si deseas truncar una tabla completa puedes usar el método `truncate`:

```
DB::table('users')->truncate();
```

Población (Seeders)

Laravel incluye la capacidad de llenar tu base de datos con datos de relleno utilizando clases de seed. Nosotros vamos a crear un seeder para la tabla de Marcas del siguiente modo:

```
php artisan make:seeder MarcaSeeder
```

Una clase de seeder solo contiene un método por defecto: run. Vamos a modificar la clase MarcaSeeder y DatabaseSeeder : [\(cod004.txt\)](#)

Ejecutando Seeders

Para ejecutar los seeders y poblar las tablas con datos, usaremos:

```
php artisan db:seed
```

IMPLEMENTAMOS EL CRUD DE MARCAS USANDO SQL EN BRUTO (RAW)

Lo primero que vamos a hacer es modificar el método index de la clase Controllers/MarcaController.php que quedará:

```
public function index()
{
    //Con RAW
    $marcas = DB::select('select id,nombre from marcas order by id;');
    // dd($marcas);
    return view('marca.lista', ['marcas' => collect($marcas)]);
}
```

Y luego la vista resources/views/marca/lista.blade.php:

```
@foreach ($marcas as $marca)
<tr>
<td>
<a href="marcas/{{ $marca->id }}/edit" class="btn btn-sm btn-warning me-1"><i class="fa fa-edit"></i>
Modif.</a>
<a href="{ route('marcas.destroy', $marca->id) }}" class="btn btn-sm btn-danger"><i class="fa fa-times"></i>
Borrar</a>
</td>
<td>{{ $marca->id }}</td>
<td>{{ $marca->nombre }}</td>
</tr>
@endforeach
```

IMPLEMENTAMOS EL CRUD DE MARCAS USANDO SQL EN BRUTO (RAW)

Ahora modificamos el método store de app/Http/Controllers/MarcaController.php:

```
public function store(Request $request)
{
    $validator = Validator::make($request->all(),
    ['nombre' => 'required|max:50|unique:marcas'],
    ['nombre.unique' => 'No se ha grabado porque la marca introducida ya la has usado antes. Introduce
otra por favor.', 'nombre.required' => 'Introduce la marca por favor.']);
    if ($validator->fails()) {
        Session::flash('danger', 'ERROR la marca no se ha podido crear.');
```

return redirect('marcas/create')->withErrors(\$validator)->withInput();

```
    }
    DB::insert('insert into marcas(nombre) values (?)', [$request->nombre]);
    Session::flash('success', 'La marca se ha creado satisfactoriamente');
    return redirect()->route('marcas.index');
}
```

Creamos la vista resources/views/marca/edit.blade.php: [\(cod005.txt\)](#)

E implementamos los métodos edit y update de app/Http/Controllers/MarcaController.php: [\(cod006.txt\)](#)

Vemos que se actualiza la marca correctamente y nos muestra los mensajes.

Para borrar un registro como el de marcas, vamos a hacerlo pidiendo la conformidad del borrado mostrando antes el registro a borrar.

Creamos la vista show.blade.php: [\(cod007.txt\)](#)

Y modificamos los métodos show y destroy de app/Http/Controllers/MarcaController.php [\(cod008.txt\)](#)

IMPLEMENTAMOS EL CRUD DE UBICACIONES USANDO QUERY BUILDER

Vamos a hacer lo mismo que hemos hecho con Marcas pero con Ubicaciones y usando Query Builder para comparar un sistema de acceso a la BBDD y otro.

Como ya tenemos el modelo creado y las rutas definidas y el CRUD va a ser “casi” igual que el de marcas, vamos a copiar la carpeta de las vistas del CRUD de marcas (resources/views/marca/) en una carpeta para las ubicaciones (resources/views/ubicacion/).

Ahora vamos copiando los métodos de la clase app/Http/Controllers/MarcaController.php en app/Http/Controllers/UbicacionController.php, haciendo los cambios oportunos, por ejemplo el método index quedará:

```
public function index()
{
    $ubicacions = DB::table('ubicacions')->get();
    return view('ubicacion.lista', ['ubicacions' => $ubicacions]);
}
```

El método créate:

```
public function create()
{
    return View('ubicacion.create');
}
```

Los métodos store, show, edit, update y destroy: [\(cod009.txt\)](#)

IMPLEMENTAMOS EL CRUD DE UBICACIONES USANDO QUERY BUILDER

Ahora vamos cambiando las vistas comenzando por resources/views/ubicacion/lista.blade.php:

[\(cod010.txt\)](#)

Y ya podemos ver la relación de ubicaciones. Continuamos con la vista ubicacion/create.blade.php:

[\(cod011.txt\)](#)

Y resources/views/ubicacion/fieldsform.blade.php:

```
<div class="form-group">
  <label for="nombre">Ubicación:</label>
  @if ($items == 'create')
    <input id="nombre" class="form-control" placeholder="Introduce la ubicación" maxlength="50"
name="nombre" type="text">
  @elseif ($items == 'edit')
    <input id="nombre" class="form-control" placeholder="Introduce la ubicación" maxlength="50"
name="nombre" type="text" value="{{ old('nombre', $ubicacion->nombre) }}">
  @else
    <input id="nombre" class="form-control" name="nombre" type="text" value="{{ old('nombre', $ubicacion-
>nombre) }}" disabled readonly>
  @endif
</div>
```

También modificamos resources/views/ubicacion/edit.blade.php: [\(cod012.txt\)](#)

Antes de probar a realizar modificaciones de ubicaciones tenemos que añadir una propiedad a la clase del modelo en (lo explicaremos más adelante) app/Models/Ubicacion.php:

IMPLEMENTAMOS EL CRUD DE UBICACIONES USANDO QUERY BUILDER

```
class Ubicacion extends Model
{
    protected $fillable = ['nombre'];
}
```

Nos queda por modificar resources/views/ubicacion/show.blade.php: [\(cod013.txt\)](#)

Y tenemos funcionando el CRUD de ubicaciones.

ELOQUENT ORM

Al usar Eloquent, cada tabla de la base de datos tiene un "Modelo" correspondiente que se utiliza para interactuar con esa tabla.

Los modelos suelen estar en el directorio `app\Models`. Vamos a usar el comando `make:model` de Artisan para generar un nuevo modelo para la tabla de ordenadores:

```
php artisan make:model Ordenador
```

Vamos a crear el controlador para ordenadores:

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

Y veamos a cómo crearíamos el modelo y controlador para softwares:

```
php artisan make:controller SoftwareController --model=Software --resource
```

Vamos a crear controladores y modelos para el resto de tablas:

```
php artisan make:controller MonitorController --model=Monitor --resource
php artisan make:controller TecladoController --model=Teclado --resource
php artisan make:controller RatonController --model=Raton --resource
php artisan make:controller ComponenteController --model=Componente --resource
php artisan make:controller ImpresoraController --model=Impresora --resource
php artisan make:controller DispRedController --model=DispRed --resource
```


ELOQUENT ORM

Para ver todos los atributos y relaciones del modelo:

```
php artisan model:show Marca
```

Por convención, se utilizará el nombre plural en "snake case" de la clase como el nombre de la tabla, a menos que se especifique otro nombre de manera explícita. Eloquent asumirá que el modelo Marca almacena registros en la tabla marcas.

```
class Marca extends Model
{
    protected $table = 'tb_marcas';
}
```

Eloquent también asumirá que la tabla de base de datos correspondiente a cada modelo tiene una columna de clave primaria denominada id, se puede cambiar:

```
class Marca extends Model
{
    protected $primaryKey = 'marca_id';
}
```

Eloquent asume que la clave primaria es un valor entero autoincremental. Si deseas usar una clave primaria no numérica o no incremental, debes definir una propiedad pública \$incrementing.

```
public $incrementing = false;
```

ELOQUENT ORM

Para obtener todos los registros de la tabla asociada al modelo:

```
use App\Models\Flight;

foreach (Flight::all() as $flight) {
    echo $flight->name;
}
```

Puedes añadir restricciones adicionales a las consultas y luego invocar el método get para recuperar los resultados:

```
$flights = Flight::where('active', 1)
    ->orderBy('name')
    ->take(10)
    ->get();

// Retrieve a model by its primary key...
$flight = Flight::find(1);

// Retrieve the first model matching the query constraints...
$flight = Flight::where('active', 1)->first();

// Alternative to retrieving the first model matching the query constraints...
$flight = Flight::firstWhere('active', 1);

$flight = Flight::findOrFail(1);
```

ELOQUENT ORM

Para insertar un nuevo registro en la base de datos:

```
$flight = new Flight;  
$flight->name = $request->name;  
$flight->save();
```

Las marcas de tiempo `created_at` y `updated_at` del modelo se establecerán automáticamente cuando se llame al método `save`. El método `save` también se puede utilizar para actualizar modelos que ya existen:

```
$flight = Flight::find(1);  
$flight->name = 'Paris to London';  
$flight->save();
```

Para eliminar un modelo, puedes llamar al método `delete`:

```
$flight = Flight::find(1);  
$flight->delete();  
$deleted = Flight::where('active', 0)->delete();
```

Si conoces la clave primaria del modelo, puedes eliminar el modelo sin recuperarlo explícitamente llamando al método `destroy`:

```
Flight::destroy(1);  
Flight::destroy(1, 2, 3);
```

DATABASE: PAGINACIÓN

El paginador de Laravel está integrado con query builder y Eloquent ORM. Hay varias formas para paginar elementos. La más simple es mediante el método `paginate`. El único argumento que se pasa al método `paginate` es el número de elementos que desea que se muestren "por página".

Veamos cómo paginamos el listado de Ubicaciones:

```
public function index()
{
    $ubicacions = DB::table('ubicacions')->paginate(10);
    return view('ubicacion.lista', ['ubicacions' => $ubicacions]);
}
```

Si sólo necesita mostrar los enlaces simples "Siguiente" y "Anterior" en su vista de paginación, puede utilizar el método `simplePaginate`:

```
$users = DB::table('users')->simplePaginate(15);
```

También se pueden paginar consultas de Eloquent:

```
$users = App\User::paginate(15);
$users = User::where('votes', '>', 100)->paginate(15);
$users = User::where('votes', '>', 100)->simplePaginate(15);
```

DATABASE: PAGINACIÓN

Las instancias de paginator son iterables y pueden ser recorridas como un array:

```
@foreach ($ubicacions as $ubicacion)
@endforeach
...
</table>
{{ $ubicacions->links() }}
```

El método links mostrará los enlaces al resto de las páginas del conjunto de resultados, Para que el HTML generado por el método links sea compatible con el Bootstrap CSS framework, hay que añadir en app/Providers/AppServiceProvider.php:

```
use Illuminate\Pagination\Paginator;

public function boot(): void
{
    Paginator::useBootstrapFive();
}
```

Gracias