

# DESARROLLO WEB EN ENTORNO SERVIDOR

**U.D. 7 (parte 2/2):  
Framework Laravel**



# Frameworks

- Conjunto estandarizado de prácticas de programación para resolver una serie de problemas habituales.
- Proporcionan una serie de clases, librerías y otros componentes para facilitar el desarrollo ágil, seguro y escalable de nuevas aplicaciones.

# Frameworks

## ■ Ventajas:

- Reutilización del trabajo ya hecho.
- Extensa documentación.
- Separación en capas.
- Seguimiento de buenas prácticas de programación.
- Escalabilidad y mantenimiento.
- Desarrollo más rápido y, por tanto, más económico.

# Frameworks

## ■ Inconvenientes:

- A veces pueden limitar el desarrollo.
- Curva de aprendizaje más costosa (más en unos que en otros).
- Dependiendo del proyecto, puede llegar a implicar más trabajo.
- Actualizaciones frecuentes, a veces imprevistas.
- Preferencias personales: Algunos programadores se sienten más cómodos si todo el código es suyo.
- Ocultan gran parte del funcionamiento de la aplicación: No son aptos para aprender a programar.

# Frameworks

- Los frameworks PHP típicamente siguen el patrón de diseño del **Modelo-Vista-Controlador** (MVC).
- Tener unos conocimientos mínimos para usar una **interfaz de línea de comandos** (CLI) ayuda cuando se usa un framework PHP.
- Por ejemplo, Laravel tiene su propia CLI, la *Consola Artisan*. Usando el comando “make” en *Artisan* se pueden construir rápidamente modelos, controladores y otros componentes para un proyecto.

# Frameworks

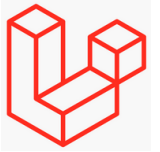
- También saber instalar por línea de comandos paquetes PHP a través de *Composer*.
- Por ejemplo, *Yii Framework* es uno de los que usa *Composer* para **instalar y administrar dependencias** o paquetes que son necesarios para que una aplicación se ejecute.
- *Packagist* es el principal **repositorio de paquetes** que se puede instalar con *Composer*.
- Algunos de los paquetes más populares de *Composer* funcionan con el framework *Symfony*.

# Frameworks

- Aquí podemos ver un listado de más de 40 frameworks de PHP: [https://en.wikipedia.org/wiki/Category:PHP\\_frameworks](https://en.wikipedia.org/wiki/Category:PHP_frameworks)  
Algunos de los más populares de hoy en día son:

- Laravel
- CodeIgniter
- Symfony
- Zend / Laminas
- Phalcon
- Yii
- CakePHP
- FuelPHP

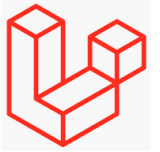




# Framework - Laravel

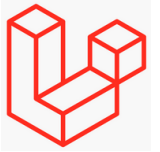
- "El framework PHP para artesanos de la web".
- Lanzamiento: Junio de 2011.
- Fue desarrollado por Taylor Otwell, quien quería un framework con elementos que CodeIgniter no tenía, como la autenticación de usuario.
- Es conocido por su elegante sintaxis que es fácil de entender y un placer para trabajar.
- Se puede comenzar a trabajar en proyectos rápidamente. También permite omitir una gran cantidad de elementos básicos, ya que se puede acceder a funciones como la autenticación de usuarios, la administración de sesiones y el almacenamiento en caché.





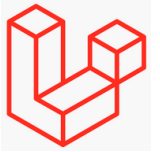
# Framework - Laravel

- Extiende la funcionalidad principal de Laravel usando extensiones.
- Integra Laravel con bibliotecas y plataformas de terceros como AWS.
- Ejecuta tareas de forma asíncrona en segundo plano para mejorar el rendimiento.
- Cuenta con una comunidad muy activa, lo que significa que no tendrás problemas para encontrar ayuda o tutoriales.
- Página oficial: <https://laravel.com/>
- Documentación: <https://documentacionlaravel.com/docs/11.x>



# Framework - Laravel

- Ventajas:
  - Sintaxis simple y elegante.
  - Mapeo objeto-relacional (ORM): Eloquent.
  - Potente sistema de plantillas para vistas: Blade.
  - Reutiliza y moderniza componentes de Symfony.
  - Es relativamente “sencillo” y potente.
  - Es previsible que domine el mercado durante los próximos años.
  - Comunidad de usuarios altamente especializada.



# Framework - Laravel

- Inconvenientes:
  - Instalación, configuración y despliegue complejos.
  - Curva de aprendizaje un poco elevada.
  - Se mueve según los intereses personales de su autor, con actualizaciones frecuentes y cambios caprichosos.
  - Inestabilidad de varios de sus componentes: A veces hay que recurrir a “fixes” o a componentes de terceros.
  - Fuerte dependencia de la consola de comandos y de herramientas de terceros (composer, vagrant, npm,...).



# Laravel - Instalación

## ■ Versiones soportadas actualmente:

VERSIÓN	FECHA DE LANZAMIENTO	CORRECCIONES DE ERRORES HASTA	ACTUALIZACIONES DE SEGURIDAD HASTA
<u>11</u>	12 de marzo de 2024	5 de agosto de 2025	3 de febrero de 2026
<u>10</u>	14 de febrero de 2023	7 de agosto de 2024	7 de febrero de 2025

## ■ Requisitos:

- PHP versión 8.2 o superior (8.3 con MAMP).
- Sistema gestor de BBDD (MySQL incluido en MAMP).
- Composer (gestor de paquetes de PHP).
- Node.js (entorno de ejecución para servidor en JS ya que Laravel utiliza un compilador JS).

## ■ Instalación: <https://documentacionlaravel.com/docs/11.x/installation>

# Laravel - Instalación

- Instalación de Composer: <https://getcomposer.org/>



A Dependency Manager for PHP

Latest: **2.8.2** ([changelog](#))

[Getting Started](#)

[Download](#)

[Documentation](#)

[Browse Packages](#)

[Issues](#)

[GitHub](#)

## Windows Installer

The installer - which requires that you have PHP already installed - will download Composer for you and set up your PATH environment variable so you can simply call `composer` from any directory.

Download and run [Composer-Setup.exe](#) - it will install the latest composer version whenever it is executed.

# Laravel - Instalación


- Instalación de Composer:
  - En la pantalla “Settings Check” hay que indicar la ruta al archivo “php.exe”, situado en el directorio “.../MAMP/bin/php/phpXX”, y marcar el checkbox “Add this PHP to your path?”.
  - En el resto de pantallas solo hay que pulsar en “Next” y en la última en “Finish”.
  - En la consola se puede comprobar que se ha instalado correctamente ejecutando el comando Composer desde cualquier ruta.

# Laravel - Instalación

- Instalación de Node.js: <https://nodejs.org/en>
  - En la pantalla “Destination Folder”, se puede cambiar la ruta a la carpeta personal.
  - En la pantalla “Tools for Native Modules” hay que marcar el checkbox “Automatically install the necessary tools”.
  - En el resto de pantallas “Next” y “Finish”.
  - Se abrirá la consola de comandos y simplemente se pulsa “Intro” para continuar.

## Run JavaScript Everywhere

Node.js® is a free, open-source, cross-platform JavaScript runtime environment that lets developers create servers, web apps, command line tools and scripts.

Download Node.js (LTS) 

# Laravel - Introducción

## ■ Creación de un proyecto:

<https://documentacionlaravel.com/docs/11.x/installation#creating-a-laravel-project>

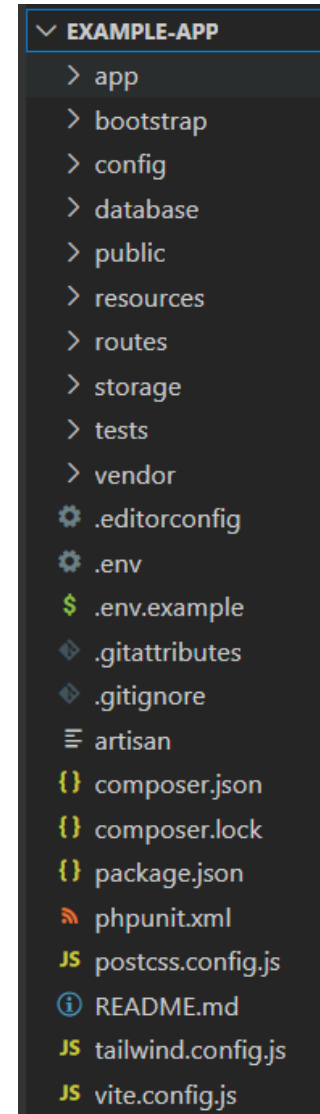
- En el archivo “.../MAMP/bin/php/phpXX/php.ini”, descomentamos las líneas “extension=fileinfo”, “extension=pdo\_mysql” y “extension=pdo\_sqlite”.
- Para crear un nuevo proyecto de Laravel debemos situarnos en el directorio de nuestro servidor local (en Apache: “.../MAMP/htdocs/...”) y abrir una terminal *Git Bash* desde donde ejecutar el comando:  
*composer create-project laravel/laravel example-app*

```
composer create-project laravel/laravel example-app --ignore-platform-req=ext-fileinfo
```



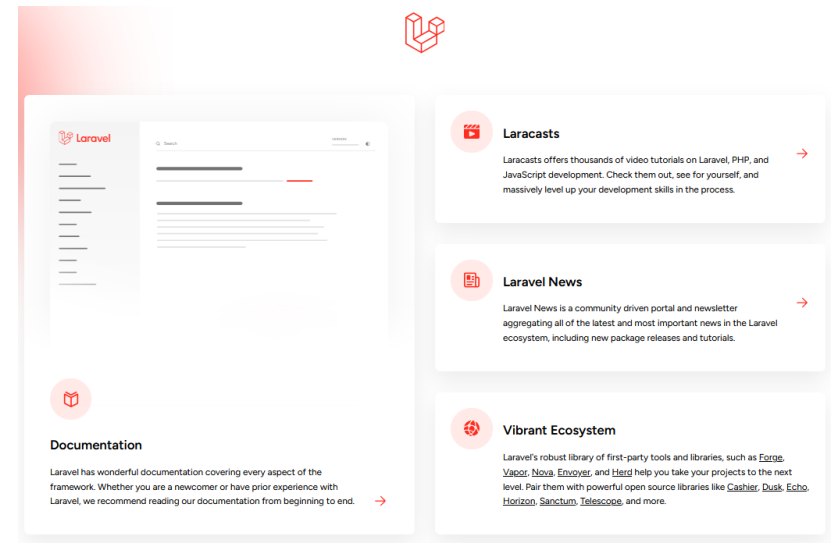
# Laravel - Introducción

- Creación de un proyecto:
  - Se creará un directorio por defecto llamado “example-app” que contendrá una nueva instalación de Laravel con todas las dependencias de Laravel ya instaladas.



# Laravel - Introducción

- Creación de un proyecto:
  - Desde la versión 11, Laravel usa por defecto *SQLite*. Lanzamos el comando de *Artisan* (desde la terminal) que instala la BD: *php artisan migrate:fresh*
  - Levantamos el servicio: *php artisan serve*
  - Abrimos en el navegador: <http://localhost:8000>



Versiones de Laravel  
y PHP instaladas

Laravel v11.30.0 (PHP v8.3.1)

# Laravel - Introducción

- Creación de un proyecto:
    - También levantando MAMP y ejecutando el archivo:  
*http://localhost/.../example-app/public/index.php*
    - Otra forma de crear nuevos proyectos en Laravel es incluyendo el instalador global de Laravel con el siguiente comando (se puede ejecutar desde cualquier directorio):  
*composer global require laravel/installer*
- Y después:
- laravel new example-app*

# Laravel - Introducción

- Estructura de directorios:
  - **composer.json**: Información para composer (administrador de paquetes de PHP). Sirve para instalar todas las librerías de terceros que Laravel necesita para funcionar.
  - **/app**: Código de nuestra aplicación (aquí están los modelos).
  - **/app/config**: Configuración de la aplicación.
  - **/app/http**: Peticiones HTTP, incluyendo los controladores.
  - **/plugins**: <https://elfsight.com/es/laravel-modules/>

# Laravel - Introducción

- Estructura de directorios:
  - **/database:** Migraciones y seeders de la BD.
  - **/public:** Directorio de acceso público. Aquí Laravel generará todo lo que hay que mover al servidor para poner la web en producción. Se encuentra el archivo “index.php”, sirve para controlar las solicitudes HTTP de la aplicación. Puedes crear aquí dentro carpetas para colocar imágenes, scripts JS o archivos CSS.
  - **/storage:** Aquí Laravel guarda su memoria caché, información sobre las sesiones, vistas compiladas... No tocar esta carpeta.

# Laravel - Introducción

- Estructura de directorios:
  - **/resources:** Aquí están las vistas. También el resto de *assets* (imágenes, CSS, JS). A diferencia del directorio “/public”, los archivos JS o CSS que se coloquen aquí estarán precompilados, no serán accesibles vía web y Laravel se encargará de compilarlos automáticamente y generar versiones minimizadas de nuestro CSS y JS. De momento, vamos a colocar nuestros CSS y JS en la carpeta “/public”.

# Laravel - Introducción

- Estructura de directorios:
  - **/vendors**: Librerías de terceros. Es importante añadir esta carpeta al “.gitignore” si vas a construir un repositorio *git* para tu aplicación Laravel, porque “/vendors” puede ocupar bastante espacio y no tiene sentido incluirla en tu proyecto. Si necesitas desplegar esta aplicación Laravel en otro servidor, basta con clonar el repositorio y ejecutar *composer update*. Eso rellenará la carpeta “/vendor” con las librerías más adecuadas para ese servidor.

# Laravel - Introducción

## ■ Convecciones:

- **Identificadores:** En inglés, mejor “User” que “Usuario”.
- **Modelos:** Igual a los de las tablas de la BD pero en singular, en *Camel/Case* empezando con mayúscula. Ejemplo: *RegisteredUser*
- **Controladores:** Como los modelos, pero añadiendo la palabra “controller”. Ejemplo: *RegisteredUserController*



# Laravel - Introducción

- **Convecciones:**
  - **Métodos:** Se nombran en *camelCase* empezando con minúscula. Ejemplo: *RegisteredUser::getAll()*
  - **Atributos:** Se nombran en *snake\_case* empezando con minúscula. Ejemplo: *RegisteredUser::first\_name*
  - **Variables:** Los identificadores deben ir en *camelCase* y empezando con minúscula. En plural si se trata de una colección y en singular si es un objeto individual o una variable simple. Ejemplo: *bannedUsers* (colección), *articleContent* (objeto individual)

# Laravel - Introducción

## ■ Convecciones:

- **Tablas:** se nombran en *snake\_case* y en plural.  
Ejemplo: *registered\_users*
- **Columnas de las tablas:** se nombran en *snake\_case*, sin referencia al nombre de la tabla.  
Ejemplo: *first\_name*
  - **Clave primaria:** Se llamarán siempre *id*, de tipo *integer* y *auto-increment*.
  - **Claves ajenas:** Se forman con el nombre de la tabla ajena en singular más la palabra *id*. Ejemplo: *article\_id*
  - **Timestamps:** Laravel siempre crea marcas de tiempo para todo. Y siempre se llaman *created\_at* y *updated\_at*, de tipo *datetime*. Acostúmbrate a tenerlas en todas tus tablas.

# Laravel - Introducción

- Variables de entorno (archivo *.env*):
  - **APP\_ENV**: En esta variable se indica si la aplicación está en desarrollo o en producción.
  - **APP\_DEBUG**: Muestra los errores para depuración. Se pone a *true* durante el desarrollo y se cambia a *false* al pasar a producción.
  - **APP\_URL**: La URL base de la aplicación.
  - **DB\_CONNECTION**, **DB\_HOST**, **DB\_USERNAME**, etc: Configuración de la conexión a la BD.

# Laravel - Introducción

- Variables de entorno (archivo `.env`):
  - Una instalación limpia de Laravel vendrá con un archivo llamado “`.env.example`”, que contiene una plantilla para que puedas construir tu propio archivo `.env`. También viene con un archivo `.env` que será el que tome por defecto para la configuración del entorno del proyecto.
  - Aviso: El archivo `.env` NO debe sincronizarse con *git* (o con el control de versiones que usemos) porque contiene información sensible. Asegúrate de incluirlo en tu “`.gitignore`”.

# Laravel - Introducción

- Variables de entorno (archivo *.env*):
  - Una vez creado nuestro archivo *.env*, podemos usar las variables definidas en él en cualquier otra parte de la aplicación. Por ejemplo, en “/config/database.php” usaremos una expresión así:  
*'default' => env('DB\_CONNECTION', 'mysql')*
  - El primer parámetro de *env()* es la variable de entorno que queremos consultar y el segundo es el valor por defecto en caso de que la variable no exista.

# Laravel - Introducción

- Variables de entorno (archivo `.env`):
  - Por defecto tiene el siguiente contenido:

The image shows a screenshot of the `.env` file in a code editor. The file contains the following configuration:

```
1 APP_NAME=Laravel
2 APP_ENV=local
3 APP_KEY=base64:oiDjzrflUz9N2Knp3eQ9yH/aRN0n8844XzyTGRX+AIs=
4 APP_DEBUG=true
5 APP_URL=http://localhost
6
7 LOG_CHANNEL=stack
8 LOG_DEPRECATIONS_CHANNEL=null
9 LOG_LEVEL=debug
10
11 DB_CONNECTION=mysql
12 DB_HOST=127.0.0.1
13 DB_PORT=3306
14 DB_DATABASE=laravel
15 DB_USERNAME=root
16 DB_PASSWORD=
```

Annotations with arrows pointing to specific lines:

- Nombre de la aplicación** (points to line 1: `APP_NAME=Laravel`)
- Entorno de la aplicación. Podría ser local, production u otro entorno que queramos crear.** (points to line 2: `APP_ENV=local`)
- Clave autogenerada al crear el proyecto.** (points to line 3: `APP_KEY=base64:oiDjzrflUz9N2Knp3eQ9yH/aRN0n8844XzyTGRX+AIs=`)
- Si tenemos activado el modo debug o no.** (points to line 4: `APP_DEBUG=true`)
- URL raíz de la aplicación.** (points to line 5: `APP_URL=http://localhost`)
- Conexión con la base de datos.** (points to lines 11-16, grouped by a bracket)

# Laravel - Introducción

- Variables de entorno (archivo `.env`):
  - El resto de variables no son de utilidad de momento. Básicamente permiten configurar la conexión con diferentes servicios como por ejemplo:
    - Almacenaje de sesiones (Redis).
    - Envío de correo electrónico.
    - Servicios en la nube (AWS).
    - Notificaciones en tiempo real (Pusher).

# Laravel - Introducción

- Configuración (archivos */config*):
  - **database.php**: Configuración de la conexión a la BD. Toma sus valores principales de *.env*, pero desde aquí se puede cambiar otras cosas, como el controlador (por defecto es MySQL).
  - **app.php**: Nombre de la aplicación, estado (desarrollo, producción,...).
  - **session.php**: Forma en la que se almacenarán las variables de sesión (por defecto en un archivo en el servidor).



# Laravel - Artisan

- Consola de comandos *Artisan*:
  - Automatiza muchas tareas habituales al trabajar con Laravel. Por ejemplo:
    - Generar esqueletos de controladores y modelos.
    - Crear migraciones de BBDD.
    - Rellenar BD con datos de prueba.
    - Hacer el enrutamiento.
    - Etc.

# Laravel - Artisan

- Ejemplo - Crear un controlador:
  - **A mano:** Vamos al directorio “/app/Http/Controllers” y creamos un archivo llamado “HolaController.php”. Luego lo rellenamos con el esqueleto de un controlador vacío, copiando de otro controlador existente y eliminando todo lo que no haga falta.
  - **Con Artisan:**  
*php artisan make:controller HolaController*  
Artisan creará directamente el archivo “/app/Http/Controllers/HolaController.php” y lo rellenará con el esqueleto de un controlador vacío sintácticamente correcto.

# Laravel - Artisan

- Comandos principales:
  - **php artisan list** → Lista todos los comandos válidos en tu instalación de Laravel.
  - **php artisan migrate** → Realiza migraciones (para crear la estructura de nuestra BD).
  - **php artisan db:seed** → Rellena de datos predefinidos nuestra BD.

# Laravel - Artisan

- Comandos principales:
  - **php artisan make:migration** → Crea una migración (para crear la estructura de la BD).
  - **php artisan make:seeder** → Crea un seeder (para rellenar con datos las tablas).
  - **php artisan make:controller** → Crea un controlador.
  - **php artisan make:model** → Crea un modelo.
  - **php artisan route:list** → Muestra todas las rutas definidas.

# Laravel - Artisan

- Crear una clave segura:

*php artisan key:generate*

- Lo primero que hay que hacer con cualquier aplicación nueva es ejecutar este comando, Laravel no funcionará sin ella.
- Esto crea una clave de encriptación aleatoria que se guarda en el archivo de configuración de la aplicación “.env”. Laravel utilizará esa clave para cifrar ciertas cosas (por ejemplo, las contraseñas de usuario).

# Laravel - Rutas

- Enrutador:
  - Es el componente que captura las URL solicitadas al servidor y las traduce a **invocaciones de métodos de los controladores**.
  - Es capaz además de **mapear fragmentos de la URL** a variables PHP que serán inyectadas como parámetros a los métodos del controlador.
  - Más en: <https://documentacionlaravel.com/docs/11.x/routing>

# Laravel - Rutas

- Enrutador (ejemplo):

- Significa que si le pides al servidor una ruta como:

*https://mi-servidor/user/delete/12*

El enrutador puede “trocear” esa URL para extraer los segmentos (“user”, “delete” y “12”) y puedes decidir qué hacer con cada uno de esos segmentos. Lo normal en este ejemplo sería que invocaras el método “delete()” del controlador “UserController”, y que ese método recibiera como parámetro el dato “12”, que será el *id* del usuario que se pretende borrar.

# Laravel - Rutas

## ■ Introducción:

- En la carpeta “/routes” se encuentran todas las rutas de la aplicación.
- El archivo por defecto para las rutas es “web.php”, que define las rutas que son para la interfaz web. A estas rutas se les asigna el grupo de *middleware web*, que proporciona características como el estado de la sesión y la protección CSRF.
- Las rutas en un archivo “api.php” no tienen estado y se les asigna el grupo de *middleware api*. Aquí es donde se definirán las rutas asociadas a una API en caso de existir.



# Laravel - Rutas

- Introducción:

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

- Aquí está la ruta que se crea por defecto en el “web.php” asociada a la URL home de la aplicación Web (/).

# Laravel - Rutas

- Crear una ruta hacia un controlador (ejemplo 1):
  - Para capturar una ruta se debe editar el archivo “/routes/web.php” y añadir el siguiente código:

```
Route::get('/hola', function() {  
    return "Hola, mundo.";  
});
```

- Cuando escribas la ruta “hola” en la barra de direcciones del navegador, se ejecutará esta función anónima, también denominadas **closure**, y como resultado, se verá el texto “Hola, mundo.” en la ventana del navegador. Pruébalo en el navegador:

*<http://localhost/.../example-app/public/index.php/hola>*

# Laravel - Rutas

- Crear una ruta hacia un controlador (ejemplo 2):
  - Los *closures* o funciones sin nombre raramente se usan en el enrutador. Lo que suele hacer el enrutador es redirigir la ejecución hacia un controlador.
  - Es lo que vamos a hacer ahora. Edita el enrutador “/routes/web.php” y sustituye la ruta anterior por:

```
Route::get('/hola', 'HolaController@index');
```

Esto indica al enrutador que, al recibir la ruta “hola”, se debe ejecutar el método “index()” del controlador “HolaController”. Pero el controlador “HolaController” no existe, así que vamos a crearlo con:

```
php artisan make:controller HolaController
```

# Laravel - Rutas

- Crear una ruta hacia un controlador (ejemplo 2):
  - Editamos el controlador “/app/Http/Controllers/HolaController.php” y le añadimos el método “index()”:

```
public function index() {  
    return "Hola, mundo";  
}
```

- Pruébalo en el navegador (no va funcionar):

*http://localhost/.../example-app/public/index.php/hola*

# Laravel - Rutas

- Crear una ruta hacia un controlador (ejemplo 2):
  - Para que este enrutamiento funcione es necesario modificar el archivo “/bootstrap/app.php” dentro del bloque *withRouting*:

```
use Illuminate\Support\Facades\Route;

return Application::configure(basePath: dirname(__DIR__))
    ->withRouting(
        web: __DIR__.'/../routes/web.php',
        commands: __DIR__.'/../routes/console.php',
        health: '/up',
        then: function () {
            Route::middleware('web')
                ->namespace('App\Http\Controllers')
                ->group(base_path('routes/web.php'));
        }
    )
```

# Laravel - Rutas

- Crear una ruta hacia un controlador (antiguo):
  - Para que este enrutamiento funcionara era necesario modificar el archivo que ya no existe  
“/app/Providers/RouteServiceProvider.php”:

```
$this->routes(function() {  
    Route::prefix('api')  
        ->middleware('api')  
        ->namespace('App\Http\Controllers') ← AÑADE ESTO  
        ->group(base_path('routes/api.php'));  
    ...  
    Route::middleware('web')  
        ->namespace($this->namespace)  
        ->namespace('App\Http\Controllers') ← AÑADE ESTO  
        ->group(base_path('routes/web.php'));  
});
```

# Laravel - Rutas

- Crear una ruta hacia un controlador (ejemplo 2):

- Actualmente, el enrutamiento se realiza mediante:

```
// Enrutador:  
Route::get('/hola', HolaController::class);  
  
// Controlador:  
public function __invoke() {  
    return "Hola, mundo";  
}
```

- Prueba de nuevo en el navegador:

*<http://localhost/.../example-app/public/index.php/hola>*

# Laravel - Rutas

- Métodos:

- Además de GET, en el enrutador se pueden enrutar otras acciones. Los métodos disponibles para las rutas son:

```
Route::get($uri, $callback);  
Route::post($uri, $callback);  
Route::put($uri, $callback);  
Route::patch($uri, $callback);  
Route::delete($uri, $callback);  
Route::options($uri, $callback);
```



# Laravel - Rutas

## ■ Métodos:

```
Route::get(); // Solicitudes habituales.  
Route::post(); // Recepción de datos de formulario (para  
    INSERT).  
Route::put(); // Recepción de datos para UPDATE (también  
    puede escribirse Route::patch(), que no es lo mismo).  
Route::delete(); // Recepción de datos para DELETE.  
Route::patch(array('GET','POST'), 'ruta', acción)  
    // Responderá tanto a GET como a POST.
```

# Laravel - Rutas

- Métodos:

- **GET:** Se utiliza para obtener recursos del servidor. Por ejemplo, si queremos obtener información de un usuario.
- **POST:** Para crear recursos en el servidor. Por ejemplo, si queremos crear un nuevo usuario en la BD.
- **PUT:** Para actualizar recursos completos en el servidor. Por ejemplo, si queremos actualizar toda la información de un usuario en la BD.

# Laravel - Rutas

- Métodos:
  - **PATCH**: Para actualizar parte de un recurso en el servidor. Por ejemplo, si queremos actualizar solamente el correo electrónico de un usuario en la BD.
  - **DELETE**: Para eliminar recursos del servidor. Por ejemplo, si queremos eliminar un usuario de la BD.
  - Los verbos **PUT**, **PATCH** y **DELETE** no están soportados aún por HTML.

# Laravel - Rutas

- Métodos:
  - Si en una misma URL queremos admitir varios métodos, podemos usar *match* especificando qué métodos admite, o *any* que los admitiría todos:

```
Route::match(['get', 'post'], '/', function () {  
    //  
});  
  
Route::any('/', function () {  
    //  
});
```

# Laravel - Rutas

- Cargar una vista desde el controlador:
  - Desde el *Route* podemos llamar a un controlador o directamente a una vista:

```
Route::view('/welcome', 'welcome');  
  
Route::view('/welcome', 'welcome', ['name' => 'Taylor']);
```

- En el primer caso, simplemente llama a la vista con el nombre “welcome”, que se encuentra en la carpeta “/resources/views” y se llama “welcome.blade.php”.
- En el segundo caso, además le pasa una variable *name* con el valor “Taylor” que podrá imprimir la vista cuando lo estime oportuno.

# Laravel - Rutas

## ■ Parámetros:

- A una ruta se le pueden pasar 0 o N parámetros y éstos podrían ser obligatorios u opcionales:

```
Route::get('/posts/{post}/comments/{comment}', function ($postId, $commentId)
    //
});
```

- La URL admite un único parámetro obligatorio “id”.

```
Route::get('/posts/{post}/comments/{comment}', function ($postId, $commentId)
    //
});
```

- La URL admite dos parámetros obligatorios “postId” y “CommentId”.

```
Route::get('/user/{name?}', function ($name = null) {
    return $name;
});

Route::get('/user/{name?}', function ($name = 'John') {
    return $name;
});
```

- La URL admite un parámetro opcional *name*. En el primer caso, por defecto se le pone el valor *null* (en caso de no recibirlo) y en el segundo se le pone un nombre por defecto, “John”.

# Laravel - Rutas

## ■ Parámetros:

- También se puede hacer que los parámetros recibidos cumplan una expresión regular:

```
Route::get('/user/{name}', function ($name) {  
    //  
})->where('name', '[A-Za-z]+');  
  
Route::get('/user/{id}', function ($id) {  
    //  
})->where('id', '[0-9]+');  
  
Route::get('/user/{id}/{name}', function ($id, $name) {  
    //  
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```

- En los ejemplos tenemos estos 3 casos:
  - Que el nombre solo acepte letras.
  - Que el “id” solo pueda ser números.
  - Ambas condiciones a la vez.

# Laravel - Rutas

- Cargar una vista desde el controlador (ejemplo 3):
  - En la arquitectura MVC, el controlador no debería producir ninguna salida HTML. Vamos a modificarlo para que el método “index()” del controlador no genere el “Hola, mundo”, sino que invoque a una vista que se encargue de ello.
  - Además, vamos a inyectar en la URL una variable con el nombre del usuario para mostrar cómo se capturan esos valores y cómo se pasan a las vistas en Laravel.



# Laravel - Rutas

- Cargar una vista desde el controlador (ejemplo 3):
  - Comenzamos modificando el enrutador “/routes/web.php”. Observa cómo se usan las llaves, { y }, para indicar la presencia de una variable en la URL:

```
Route::get('/hola/{nombre}', 'HolaController@show');
```

# Laravel - Rutas

- Cargar una vista desde el controlador (ejemplo 3):
  - Ahora creamos un método “show()” en el controlador “/app/Http/Controllers/HolaController.php”. Estamos invocando una vista llamada “hola” y le estamos pasando un *array* con los datos necesarios (el nombre del usuario, en este caso):

```
public function show($nombre) {  
    $data['nombre'] = $nombre;  
    return view('hola', $data);  
}
```

# Laravel - Rutas

- Cargar una vista desde el controlador (ejemplo 3):
  - Esa vista debe crearse en un archivo nuevo “/resources/views/hola.blade.php” y puede tener este aspecto:

```
<body>
    Saludos, {{$nombre}}.
    Ha ingresado en el sitio web DWES.
</body>
```

- Puedes probar esta nueva ruta cargando en el navegador una ruta como:

*http://localhost/.../example-app/public/index.php/hola/ProfesorFalken*

# Laravel - Rutas

- Enrutamiento básico:
  - Como hemos visto en el ejemplo “Hola, mundo”, hay varias formas de generar una salida HTML desde el enrutador “/routes/web.php”. En este código de ejemplo vemos las cuatro formas básicas:

```
// Forma 1: Generar la salida directamente en el
enrutador, con un closure (función sin nombre):
Route::get('/hola', function() {
    return "Hola, mundo";
});
```

```
// Forma 2: Llamar a una función de un controlador
sin pasarle parámetros:
Route::get('/hola', 'HolaController@show');
```

# Laravel - Rutas

- Enrutamiento básico:

```
// Forma 3: Llamar a una función de un controlador  
pasándole parámetros:
```

```
Route::get('/hola/{nombre}', 'HolaController@show');
```

```
// Forma 4: Llamar a una función de un controlador  
con un parámetro optativo:
```

```
Route::get('/hola/{nombre?}', 'HolaController@show');
```

# Laravel - Rutas

- Enrutamiento básico:
  - La diferencia entre la forma 3 y la 4 es que, en la 3, la ruta debe llevar forzosamente un dato a continuación de “/hola”, (algo como “https://mi-servidor/hola/juan”). Si no lo lleva, el enrutador considerará que no se trata de esa ruta y seguirá buscando alguna ruta coincidente en el resto del archivo.
  - En cambio, en la 4, el dato final es optativo, así que el enrutador invocará el método “show()” del controlador tanto si ese dato aparece en la URL como si no lo hace.

# Laravel - Rutas

- Nombres:
  - Una opción interesante que tienen las rutas de Laravel, es ponerle un nombre **name**. Permite utilizar este nombre cuando se enlaza desde otra parte de la aplicación y si se tuviera que modificar la ruta, no habría que modificar todos los archivos donde se utiliza:

```
Route::get('/user/profile', function () {  
    //  
})->name('profile');
```

```
Route::get(  
    '/user/profile',  
    [UserProfileController::class, 'show']  
)->name('profile');
```

Se puede poner el nombre tanto en una URL simple como en una que use un controlador.

# Laravel - Rutas

## ■ Nombres:

- Es recomendable asignar un nombre a las rutas en el enrutador. Esto hace que, más adelante, podamos cambiar la URL de los enlaces sin tener que modificar el código fuente de nuestras vistas.
- El nombre se le asigna añadiendo `->name('nombre')` al final:

```
Route::get('/contactar',  
    'ContactController@contact')->name('contact')
```



# Laravel - Rutas

- Nombres:
  - En tu código fuente, debes referirte a esta ruta siempre con la expresión `route('contact')` (ya veremos exactamente cómo se hace esto), pero el usuario verá la dirección “https://servidor/contactar”.
  - En el futuro se puede cambiar la forma en la que lo ve el usuario. Por ejemplo, puedes cambiar `Route::get('/contactar'...)` por `Route::get('/acerca-de'...)`, pero no tendrás que modificar ni una línea de código más en tu aplicación, porque internamente esa ruta seguirá llamándose `route('contact')`.

# Laravel - Rutas

## ■ Nombres:

- Desde un archivo PHP llamamos a la ruta por el nombre.
- Llamamos a la ruta por el nombre y le pasamos el parámetro “id”.
- Llamamos a la ruta por el nombre y le pasamos los parámetros “id” y “photos”.

```
// Generating URLs...  
$url = route('profile');  
  
// Generating Redirects...  
return redirect()->route('profile');  
  
return to_route('profile');
```

```
Route::get('/user/{id}/profile', function ($id) {  
    //  
})->name('profile');  
  
$url = route('profile', ['id' => 1]);
```

```
Route::get('/user/{id}/profile', function ($id) {  
    //  
})->name('profile');  
  
$url = route('profile', ['id' => 1, 'photos' => 'yes']);  
  
// /user/1/profile?photos=yes
```

# Laravel - Rutas

## ■ Grupos:

- Otra opción interesante es la opción de agrupar URLs, creando **groups**, que compartan parte de la URL o utilicen el mismo *middleware*, por ejemplo:

```
Route::middleware(['first', 'second'])->group(function () {  
    Route::get('/', function () {  
        // Uses first & second middleware...  
    });  
  
    Route::get('/user/profile', function () {  
        // Uses first & second middleware...  
    });  
});
```

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

- Creamos un *group* con dos URLs que comparten los *middlewares first y second*.
- Creamos un *group* con las URLs que comparten el mismo controlador.

# Laravel - Rutas

## ■ Grupos:

```
Route::domain('{account}.example.com')->group(function () {  
    Route::get('user/{id}', function ($account, $id) {  
        //  
    });  
});
```

- Creamos un *group* con las URLs que comparten un subdominio.

```
Route::name('admin.')->group(function () {  
    Route::get('/users', function () {  
        // Route assigned name "admin.users"...  
    })->name('users');  
});
```

```
Route::prefix('admin')->group(function () {  
    Route::get('/users', function () {  
        // Matches The "/admin/users" URL  
    });  
});
```

- Creamos un *group* con las URLs que comparten el prefijo “admin”, es decir, todas las rutas tendrán el formato “/admin”.

- Creamos un *group* con las URLs que comparten el nombre “admin”, es decir, todas las rutas tendrán el nombre “admin.XX”.

# Laravel - Rutas

## ■ Grupos:

- Por último, como parámetro, además de enviarle variables simples, también se le pueden pasar objetos (modelos) de la propia aplicación:

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

- Al pasarle el objeto “User” completo, podríamos acceder directamente a alguno de sus atributos como, por ejemplo, el “email”.

# Laravel - Rutas

## ■ Orden:

- El orden en el que se escriben las rutas en el enrutador es importante.
- Por ejemplo, si pedimos la dirección “http://mi-servidor/usuario/crear”, escribir estas dos rutas en este orden es un error:

```
Route::get('usuario/{nombre}', 'UsuarioController@show');  
Route::get('usuario/crear', 'UsuarioController@create');  
// El enrutador tratará de mostrar un usuario cuyo nombre  
sea “crear” (que seguramente no existirá) porque la  
petición encaja con las dos rutas y el enrutador elegirá  
la primera ruta que encuentre.
```

# Laravel - Rutas

## ■ Orden:

- La solución pasa por alterar el orden de las líneas en el enrutador.
- De este modo, la petición “http://mi-servidor/usuario/crear” seguirá encajando en las dos rutas, pero el enrutador elegirá la 1ª.
- En cambio, una petición parecida (por ejemplo “http://mi-servidor/usuario/luis”), solo encajará con la 2ª:

```
Route::get('usuario/crear', 'UsuarioController@create');  
Route::get('usuario/{nombre}', 'UsuarioController@show');
```

# Laravel - Rutas

- Redirecciones:
  - También podemos establecer redirecciones: De una URI a otra, devolviendo un código de error o una redirección permanente.

```
Route::redirect('/here', '/there');
```

```
Route::redirect('/here', '/there', 301);
```

```
Route::permanentRedirect('/here', '/there');
```



# Laravel - Rutas

## ■ Formularios:

- No se puede crear un formulario así: `<form method='PUT'>`, porque el navegador no lo entenderá. Solo puedes poner `<form method='GET'>` o `<form method='POST'>`.
- Se puede emular PUT, PATCH o DELETE en los formularios así:

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

Ejemplo: <https://laracoding.com/sending-a-form-to-a-put-route-in-laravel-with-example/>

# Laravel - Servidor RESTFUL

## ■ Introducción:

- Un servidor RESTful es aquel que responde a la **arquitectura REST**.
- La arquitectura REST es una forma estandarizada de construir un servidor para que realice las tareas típicas de mantenimiento de recursos. Y los recursos pueden ser cualquier cosa que se almacene en el servidor: usuarios, clientes, productos, facturas,...
- Es decir, el 99% de las veces, los recursos son registros en una tabla de la BD.

# Laravel - Servidor RESTFUL

- Rutas:
  - El enrutador de un servidor RESTful contendrá las 7 **operaciones** definidas en la arquitectura REST para cada recurso accesible desde la red y que permiten manipular el recurso:
    - Mostrarlo (un único recurso o todos)
    - Buscarlo (un único recurso o todos)
    - Insertarlo
    - Modificarlo
    - Borrarlo

# Laravel - Servidor RESTFUL

## ■ Rutas (ejemplo):

- Para un recurso llamado “user”, esas 7 operaciones son:

```
Route::get('user', 'UserController@index')->name('user.index'); //
Recupera todos los usuarios.
Route::get('user/{user}', 'UserController@show')->name('user.show'); //
Recupera usuario con id=user.
Route::get('user/crear', 'UserController@create')->name('user.create');
// Lanza el formulario de creación de usuarios.
Route::post('user/{user}', 'UserController@store')->name('user.store');
// Recoge los datos del formulario y los inserta en la BD.
Route::get('user/{user}/edit', 'UserController@edit')->name('user.edit');
// Lanza el formulario de modificación de usuarios.
Route::patch('user/{user}', 'UserController@update')->
>name('user.update'); // Recoge los datos del formulario y modifica el
usuario de la BD.
Route::delete('user/{user}', 'UserController@destroy')->
>name('user.destroy'); // Elimina al usuario de la BD.
```

# Laravel - Servidor RESTFUL

## ■ Rutas:

- Si estás construyendo un servidor RESTful debes respetar escrupulosamente los nombres y URLs de las rutas. Así, cualquier otro usuario o aplicación que use tu servidor sabrá cómo manipular los recursos sin necesidad de consultar la documentación.
- Laravel permite resumir esas entradas del enrutador en una sola línea que engloba a las 7 rutas REST:

```
Route::resource('user');
```

# Laravel - Middleware

## ■ Introducción:

- Es un mecanismo para **filtrar las solicitudes HTTP** que llegan a la aplicación. Literalmente, se ponen “en medio” de cualquier petición al servidor.
- Un **ejemplo** de middleware que tiene Laravel por defecto es el que verifica que el usuario de la aplicación esté autenticado (auth): Si el usuario no está autenticado, el *middleware* redirigirá al usuario a la pantalla de inicio de sesión. Sin embargo, si está autenticado, el *middleware* permitirá que la solicitud continúe en la aplicación.
- Más en: <https://documentacionlaravel.com/docs/11.x/middleware>

# Laravel - Middleware

## ■ Ejemplo:

- Para crear un nuevo *middleware*, se debe lanzar el comando de *Artisan*:

```
php artisan make:middleware EnsureTokenIsValid
```

- Este comando creará la clase *EnsureTokenIsValid* dentro de la ruta “/app/Http/Middleware” donde podremos hacer nuestras comprobaciones:
  - Tiene un método “handle” que es el que captura la llamada al *middleware*.
  - Como parámetros recibe *\$request* que son las variables enviadas por POST y el *\$next*, que permite continuar a la siguiente ejecución.

# Laravel - Middleware

- Ejemplo:
  - En el ejemplo:
    - Se hace la comprobación de que el *token* recibido es distinto del que queremos redirige a la “home”.
    - En caso de que sea igual, permite continuar, hacia otro *middleware* o a la siguiente ejecución que tenga la ruta enviando la variable *\$request*.

```
namespace App\Http\Middleware;

use Closure;

class EnsureTokenIsValid
{
    /**
     * Handle an incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if ($request->input('token') !== 'my-secret-token') {
            return redirect('home');
        }

        return $next($request);
    }
}
```



# Laravel - Middleware

- Rutas (ejemplo):
  - Además de generar el *middleware*, debemos añadirlo en el listado de *middlewares* que utiliza la aplicación. Para ello, accedemos al archivo “/bootstrap/app.php”:
    - Dentro del bloque *withMiddleware*, debemos añadir una fila nueva con el *middleware* **global** que hemos creado y le asociamos el nombre con el que luego le llamaremos desde el archivo de rutas:

```
use App\Http\Middleware\EnsureTokenIsValid;  
  
->withMiddleware(function (Middleware $middleware) {  
    $middleware->append(EnsureTokenIsValid::class);  
})
```

# Laravel - Middleware

- Rutas (ejemplo):
  - Cuando queremos que una ruta cumpla el *middleware*, podemos ponerle directamente el nombre (alias) que acabamos de crear:

```
->withMiddleware(function (Middleware $middleware) {  
    $middleware->append(EnsureTokenIsValid::class);  
  
    Route::get('/profile', function () {  
        //  
    })->middleware(EnsureTokenIsValid::class);
```

# Laravel - Middleware

## ■ Rutas (antiguo):

- Además de generar el *middleware*, debemos añadirlo en el listado de *middlewares* que utiliza la aplicación. Para ello, accedemos al archivo “Kernel.php” que se encuentra en la ruta “/app/Http”:

```
protected $routeMiddleware = [  
    'auth' => \App\Http\Middleware\Authenticate::class,  
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,  
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,  
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,  
    'can' => \Illuminate\Auth\Middleware\Authorize::class,  
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,  
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,  
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,  
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,  
];
```

- Dentro del bloque *\$routeMiddleware*, debemos añadir una fila nueva con el *middleware* que hemos creado y le asociamos el nombre con el que luego le llamaremos desde el archivo de rutas.

# Laravel - Middleware

## ■ Rutas (antiguo):

```
use App\Http\Middleware\EnsureTokenIsValid;

Route::get('/profile', function () {
    //
})->middleware(EnsureTokenIsValid::class);
```

- Cuando queremos que una ruta cumpla el *middleware*, podemos ponerle directamente el nombre (alias) que acabamos de crear en el archivo “Kernel.php”.

# Laravel - Middleware

- Grupos (ejemplo):
  - También se pueden crear grupos de *middlewares*. Para ello, accedemos al archivo “/bootstrap/app.php” dentro del bloque *withMiddleware*:
    - A veces, se puede querer agrupar varios *middleware* bajo una sola clave para hacer que sea más fácil asignarlos a rutas. Los grupos de *middleware* pueden asignarse a rutas y acciones del controlador usando la misma sintaxis que los *middleware* individuales:

```
Route::get('/', function () {  
    //  
})->middleware('web');  
Route::middleware(['web'])->group(function () {  
    //  
});
```

# Laravel - Middleware

- Grupos (antiguo):
  - También se pueden crear grupos de *middlewares*. Para ello, accedemos al archivo “Kernel.php” que se encuentra en la ruta “/app/Http”:

```
protected $middlewareGroups = [  
    'web' => [  
        \App\Http\Middleware\EncryptCookies::class,  
        \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,  
        \Illuminate\Session\Middleware\StartSession::class,  
        \Illuminate\View\Middleware\ShareErrorsFromSession::class,  
        \App\Http\Middleware\VerifyCsrfToken::class,  
        \Illuminate\Routing\Middleware\SubstituteBindings::class,  
    ],  
    'api' => [  
        'throttle:api',  
        \Illuminate\Routing\Middleware\SubstituteBindings::class,  
    ],  
];
```

- Dentro del bloque `$middlewareGroups`, debemos añadir una fila nueva con el *middleware* que hemos creado y pertenecería al grupo con el nombre asociado.

# Laravel - Middleware

## ■ Grupos (antiguo):

```
Route::get('/', function () {  
    //  
})->middleware('web');  
  
Route::middleware(['web'])->group(function () {  
    //  
});
```

- Cuando queremos que una ruta cumpla un grupo de *middlewares*, podemos ponerle directamente el nombre del grupo o pasarle varios grupos incluso.

# Laravel - Middleware

- Parámetros (ejemplo):
  - Por último, también puede ser interesante enviarles parámetros a los *middlewares*. Para ello, en la clase del *middleware* que creemos, le añadimos un parámetro extra:
    - Le pasamos el parámetro *\$role* para hacer ciertas comprobaciones en el *middleware*.

```
namespace App\Http\Middleware;

use Closure;

class EnsureUserHasRole
{
    /**
     * Handle the incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @param string $role
     * @return mixed
     */
    public function handle($request, Closure $next, $role)
    {
        if (! $request->user()->hasRole($role)) {
            // Redirect...
        }

        return $next($request);
    }
}
```



# Laravel - Middleware

- Parámetros (ejemplo):
  - Desde la ruta, le pasamos el parámetro después de los dos puntos (:). En este caso, le decimos que el rol es “editor”:

```
Route::put('/post/{id}', function ($id) {  
    //  
})->middleware('role:editor');
```

# Laravel - Middleware

- Ejercicio resuelto 1 (solo teoría): Crear dos *middlewares*.

Uno que compruebe si el rol del usuario es “superadmin” y el otro que compruebe si su correo es “@campusviu.es”.

En caso de no cumplirse alguno de ellos, redirigir a una pantalla de error. Si se cumplen ambos, llevar a la pantalla de “bienvenida”.

# Laravel - Middleware

- Ejercicio resuelto 1 (solución):

***Creamos los dos middlewares a través comandos de Artisan:***

```
php artisan make:middleware CheckRole  
php artisan make:middleware CheckMail
```

***Rellenamos la funcionalidad en las clases middlewares que nos ha creado:***

```
namespace App\Http\Middleware;  
  
use Closure;  
  
class CheckRole  
{  
    /**  
     * Handle an incoming request.  
     *  
     * @param \Illuminate\Http\Request $request  
     * @param \Closure $next  
     * @return mixed  
     */  
    public function handle($request, Closure $next, $role)  
    {  
        if (!$request->user()->hasRole($role)) {  
            return redirect('error');  
        }  
  
        return $next($request);  
    }  
}
```

# Laravel - Middleware

- Ejercicio resuelto 1 (solución):

*Rellenamos la funcionalidad en las clases middlewares que nos ha creado:*

```
namespace App\Http\Middleware;

use Closure;

class CheckMail
{
    /**
     * Handle an incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        if (!str_contains($request->user()->mail, '@campusviu.es')) {
            return redirect('error');
        }

        return $next($request);
    }
}
```

# Laravel - Middleware

- Ejercicio resuelto 1 (solución):

*Los añadimos en el fichero “app.php” y le asociamos un nombre:*

```
use App\Http\Middleware\CheckRole;  
use App\Http\Middleware\CheckMail;  
  
$middleware->append(CheckRole::class);  
$middleware->append(CheckMail::class);
```

# Laravel - Middleware

- Ejercicio resuelto 1 (antiguo):

*Los añadiríamos en el fichero “Kernel.php” y le asociaríamos un nombre:*

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' => \Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'bindings' => \Illuminate\Routing\Middleware\SubstituteBindings::class,
    'cache.headers' => \Illuminate\Http\Middleware\SetCacheHeaders::class,
    'can' => \Illuminate\Auth\Middleware\Authorize::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'password.confirm' => \Illuminate\Auth\Middleware\RequirePassword::class,
    'signed' => \Illuminate\Routing\Middleware\ValidateSignature::class,
    'throttle' => \Illuminate\Routing\Middleware\ThrottleRequests::class,
    'verified' => \Illuminate\Auth\Middleware\EnsureEmailIsVerified::class,
    'check-role' => \App\Http\Middleware\CheckRole::class,
    'check-mail' => \App\Http\Middleware\CheckMail::class,
];
```

# Laravel - Middleware

- Ejercicio resuelto 1 (solución):

*Añadimos en “web.php” la ruta que llama a los middlewares y creamos la de error y home, a las que se redirige en caso de éxito o fallo:*

```
Route::get('/', function () {  
    return view('welcome');  
})->name('home');  
  
Route::get('/error', function () {  
    return view('error');  
})->name('error');  
  
Route::get('/test', function () {  
    return redirect()->route('home');  
})->middleware(['check-role:superadmin', 'check-mail']);
```

# Laravel - Controladores

## ■ Introducción:

- Son un elemento clave de la arquitectura MVC. En Laravel, los controladores funcionan exactamente igual que en las aplicaciones MVC hechas con PHP nativo. Es decir, son los **puntos de entrada a la aplicación desde el enrutador**.
- Los controladores deberían permanecer lo más sencillos posible: Nada de accesos a la BD ni generación de HTML. Esas acciones se derivarán a los modelos y las vistas. El controlador es un organizador del flujo de la aplicación: Decide qué componente tiene que trabajar y el orden en el que lo debe hacer.



# Laravel - Controladores

## ■ Introducción:

- Siempre heredan de la clase *Controller* o de una subclase suya.
- Su nombre debería escribirse en singular “CamelCase” y terminando en la palabra “Controller: UserController, LoginController,...”.
- Cada método del controlador debe terminar con un *return*. Lo que el método devuelva será convertido automáticamente en una “HTTP response 200”, es decir, en una respuesta válida HTTP, excepto si es un *array*, en cuyo caso Laravel lo convertirá automáticamente en *JSON*.
- Más en: <https://documentacionlaravel.com/docs/11.x/controllers>

# Laravel - Controladores

- Introducción:
  - Se encuentran en la ruta “app/Http/Controllers” y siguen la estructura siguiente:
    - Extiende la clase *Controller* y tiene un método “show()” que recibe como parámetro un *\$id* y lo que hace es buscar el usuario con el *id* recibido y devuelve el objeto *usuario* a la vista *profile*.

```
namespace App\Http\Controllers;

use App\Models\User;

class UserController extends Controller
{
    /**
     * Show the profile for a given user.
     *
     * @param int $id
     * @return \Illuminate\View\View
     */
    public function show($id)
    {
        return view('user.profile', [
            'user' => User::findOrFail($id)
        ]);
    }
}
```

# Laravel - Controladores

- Rutas:
  - Para indicar desde una ruta “web.php” que se acceda a un método de un controlador:

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

# Laravel - Controladores

- Creación:
  - Se pueden crear a mano: En el directorio “/app/http/controllers”, creas allí un archivo vacío y empiezas a escribir código. Pero nadie lo hace así porque *Artisan* ya crea el esqueleto del archivo por tí. Mejor abrir una consola en tu servidor web y escribir.
  - Forma 1. Crear un controlador vacío:

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

# Laravel - Controladores

- Creación:

- Forma 2. Crear un controlador de tipo *resource*.

Estos controladores se generan automáticamente con un andamiaje para construir recursos REST. Es decir, la clase ya llevará incorporados los métodos “index(), create(), store(), show(), edit(), update() y destroy()” del estándar REST.

Para crear un controlador RESTful:

```
$ php artisan make:controller UserController --resource
```

# Laravel - Controladores

- Creación:
  - Forma 2. Crear un controlador de tipo *resource*.

No te olvides de añadir al enrutador “/routes/web.php” las rutas REST para este tipo controlador. Se pueden resumir las 7 rutas en esta sola entrada del enrutador:

```
Route::resource('nombreRecurso', 'controlador');
```

En el ejemplo:

```
Route::resource('usuarios', 'UserController');
```

# Laravel - Controladores

- Creación:

- Forma 3. Crear un controlador de tipo *API*.

Una API (Application Programming Interface) es un interfaz entre programas. Es decir, es la forma en la que unos interaccionan con otros.

Algunas aplicaciones web se diseñan para que otros programas las utilicen, no para que las utilicen seres humanos. En estos casos, la interfaz de usuario no existe (o es mínima) y lo importante es la API. Y los métodos del controlador no devuelven vistas, sino datos formateados en JSON.

# Laravel - Controladores

## ■ Creación:

- Forma 3. Crear un controlador de tipo *API*.

Se puede construir un controlador tipo *API* de forma muy simple, porque es parecido a un *resource* pero sin “create()” ni “edit()”, porque una *API* no necesita mostrar los formularios de inserción/modificación:

```
$ php artisan make:controller UserController --api
```

No olvidar las entradas en el enrutador. Se pueden englobar todas en una sola entrada así:

```
Route::apiResource('usuarios', 'UserController');
```



# Laravel - Controladores

- Invocables:
  - También pueden existir controladores con un único método “\_\_invoke()”. Estos son controladores invocables, que simplemente **ejecutarán una única acción**. Para crearlo ejecutamos el comando:

```
php artisan make:controller ProvisionServer --invokable
```

```
class ProvisionServer extends Controller
{
    /**
     * Provision a new web server.
     *
     * @return \Illuminate\Http\Response
     */
    public function __invoke()
    {
        // ...
    }
}
```

# Laravel - Controladores

- Invocables:
  - Y desde la ruta lo llamaríamos así:

```
use App\Http\Controllers\ProvisionServer;  
  
Route::post('/server', ProvisionServer::class);
```

# Laravel - Vistas

## ■ Introducción:

- Las vistas de Laravel se encuentran en la ruta “resources/views” y utilizan *Blade*.
- Ejemplo:

```
<!-- View stored in resources/views/greeting.blade.php -->

<html>
  <body>
    <h1>Hello, {{ $name }}</h1>
  </body>
</html>
```

El contenido entre las dobles llaves `{{ $name }}` es reemplazado por `<?php echo $name; ?>`

- Más en: <https://documentacionlaravel.com/docs/11.x/views>

# Laravel - Vistas

## ■ Ejemplo:

- Para devolver desde una ruta a una vista directamente pasándole parámetros:

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

- Para llamar a una vista desde un controlador pasándole parámetros:

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

- También se le pueden pasar así:

```
return view('greeting')  
    ->with('name', 'Victoria')  
    ->with('occupation', 'Astronaut');
```

# Laravel - Blade

- Introducción:
  - Es un **motor de plantillas** simple pero potente.
  - A diferencia de otros motores de plantillas PHP, *Blade* no impide usar código PHP básico en sus vistas. De hecho, todas las vistas de *Blade* se compilan en código PHP nativo y se almacenan en caché hasta que se modifican.
  - Dos de los principales beneficios de usar *Blade* son la **herencia de plantillas** y las **secciones**. Permitirá generar plantillas de vistas para minimizar el código que necesitaremos para nuestras vistas.

# Laravel - Blade

## ■ Introducción:

- Las plantillas *Blade* admiten condiciones y bucles para operar con las variables PHP, de modo que la misma plantilla se comporta de forma diferente con diferentes conjuntos de datos. ¡Se acabó la pesadilla de abrir y cerrar comillas en las instrucciones *echo*!. Tampoco será necesario abrir y cerrar php `<?php ... ?>` para operar con las variables del servidor y generar la salida.
- El código escrito con *Blade* no solo es más limpio y fácil de depurar que con PHP básico, sino también más seguro porque impide cualquier ataque con XSS.
- Más en: <https://documentacionlaravel.com/docs/11.x/blade>

# Laravel - Blade

- Layout:
  - El **master layout** es el diseño maestro del que derivan todas las vistas. Las aplicaciones web suelen tener uno muy definido y todas las vistas de la aplicación lo respetan.
  - Suele almacenarse por convenio en el archivo `“/resources/views/layouts/app.blade.php”` o `“/resources/views/master.blade.php”`.

# Laravel - Blade

- Layout:
  - Primero, examinaremos un diseño de página “maestro” (layout). 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 *Blade*.

```
<!-- resources/views/layouts/app.blade.php -->

<html>
  <head>
    <title>App Name - @yield('title')</title>
  </head>
  <body>
    @section('sidebar')
      This is the master sidebar.
    @show

    <div class="container">
      @yield('content')
    </div>
  </body>
</html>
```



# Laravel - Blade

- Layout (ejemplo):

```
<html>
  <head> <title>@yield('title')</title> </head>
  <body>
    @section('sidebar')
      Este es mi master sidebar.
    @show
    <div class="container">
      @yield('content')
    </div>
  </body>
</html>
```

# Laravel - Blade

## ■ Layout:

- La directiva `@section` define una sección de contenido.
- La directiva `@yield` para mostrar el contenido de una sección determinada.
- `@show` define y muestra el contenido de la sección.
- Al crear otra vista (secundaria), se debe usar la directiva `@extends` para especificar qué diseño (layout) debe “heredar” la vista secundaria.
- Las vistas que amplían un *layout* de *Blade* pueden inyectar contenido en las secciones del *layout* mediante directivas `@section`.

# Laravel - Blade

## ■ Layout:

- El contenido de estas secciones se mostrará en el *layout* en la sección *@yield*:

```
<!-- resources/views/child.blade.php -->

@extends('layouts.app')

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

@section('sidebar')
    @parent

    <p>This is appended to the master sidebar.</p>
@endsection

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

- Se puede hacer uso de la directiva *@parent* para añadir contenido a una sección en lugar de reemplazarlo.
- Para indicar que se termina el contenido a incluir en una sección se usa *@endsection*.

# Laravel - Blade

## ■ Layout:

- La directiva `@yield` también acepta un segundo parámetro donde se le indique la vista a mostrar:

```
@yield('content', 'Default content')
```

- También se puede devolver una vista secundaria desde las rutas directamente:

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

# Laravel - Blade

- Layout (ejemplo):
  - Después de crear el *master layout*, lo siguiente es construir vistas que lo utilicen. Es decir, que hereden de él todos sus componentes y añadan nuevos elementos, respetando la configuración básica establecida por el *master layout*.

```
@extends('master')
@section('title', 'Titulo de la página')
@section('sidebar')
    <p>Esto se añadirá al master sidebar.</p>
@endsection
@section('content')
    <p>Aquí va el contenido de mi página.</p>
@endsection
```

# Laravel - Blade

## ■ Layout:

- `@extends('master')` → Indica que la vista hereda del master layout.
- `@section('title', 'Título de la página')` → Aquí se define el contenido de la sección 'title' que en el master layout estaba vacía.
- `@section('sidebar')` → Aquí añade contenido a la sección 'sidebar', que en el master layout no estaba vacía.
- `@endsection` → Se emplea cuando el contenido de una sección ocupa varias líneas de código y no es posible escribirlo en la propia directiva.

# Laravel - Blade

## ■ Variables:

- Para mostrar contenido dinámicamente en una vista con Blade, debemos enviar las variables empaquetadas en un *array* (hay varias maneras):

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

- Y en la vista, imprimiríamos el contenido de la variable *name* de la siguiente manera:

```
Hello, {{ $name }}.
```

# Laravel - Blade

- Estructuras:
  - Blade admite expresiones condicionales para modificar el aspecto de una vista dependiendo del valor de una variable o del estado de la aplicación.
  - Por supuesto, eso también puede hacerse sin Blade: solo hay que usar un *if* de PHP. Pero con Blade es más fácil y limpio porque no tendremos que andar abriendo y cerrando PHP ni peleándonos con las comillas del *echo*.



# Laravel - Blade

- Estructuras - Condicionales:
  - Para los condicionales *if* o *if-else*, existen las directivas:

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

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

- También hay directivas abreviadas para el *if isset* o *if isempty*:

```
@isset($records)
    // $records is defined and is not null...
@endisset

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

# Laravel - Blade

- Estructuras - Switch:
  - Directivas del *switch*:

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

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

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

# Laravel - Blade

- Estructuras - Bucles:
  - Directivas de los *bucles*:

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

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

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

```
@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
```

# Laravel - Blade

- Estructuras - Bucles:
  - Dentro de un bucle, existe una variable *\$loop* que proporciona acceso a algunos bits de información útiles, como el índice del ciclo actual y si ésta es la primera o la última iteración del ciclo:

```
@foreach ($users as $user)
    @if ($loop->first)
        This is the first iteration.
    @endif

    @if ($loop->last)
        This is the last iteration.
    @endif

    <p>This is user {{ $user->id }}</p>
@endforeach
```

# Laravel - Blade

- Inicio de sesión:
  - También existen directivas para comprobar si el usuario está “logueado” o no:

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

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

```
@auth('admin')
    // The user is authenticated...
@endauth

@guest('admin')
    // The user is not authenticated...
@endguest
```

# Laravel - Blade

## ■ Comentarios y PHP:

- Para escribir comentarios en Blade se utilizará `{{-- --}}`:

```
{{-- This comment will not be present in the rendered HTML --}}
```

- También existe la directiva `@php` para incluir código específico de PHP como la declaración de una nueva variable o realizar algún tipo de operación con variables:

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

# Laravel - Blade

- Formularios (ejemplo):
  - Blade también facilita el tratamiento de los formularios. Además, filtra por nosotros cualquier código malicioso que traten de colarnos a través de ellos:

```
<form method="POST" action="{{ route('mi-ruta') }}">
    <!-- Ruta por su nombre -->
    @csrf <!-- Para evitar ataques CSRF -->
    <input type="email" name="email"/><br/>
    <input type="text" name="asunto"/><br/>
    <textarea name="contenido"></textarea><br/>
    <button type="submit">Enviar</button>
</form>
```

# Laravel - Blade

## ■ Formularios:

- Cuando se utiliza el método POST, se debe incluir un campo de *token CSRF* (método por el cual un usuario malintencionado intenta hacer que los usuarios, sin saberlo, envíen datos que no quieren enviar) oculto en el formulario para que el *middleware* de protección CSRF pueda validar la solicitud. Para esto se utiliza la directiva `@csrf`, para generar el *token* aleatorio que se comprobará cuando los datos regresen al servidor:

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



# Laravel - Blade

- Formularios:
  - Si en un formulario vamos a hacer uso de los métodos PUT, PATCH o DELETE debemos incluir la directiva *@method* con el método correspondiente (no existen en HTML, Laravel los simula con un campo oculto). Además, la ruta a la que apunte el formulario deberá aceptar este método:

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

# Laravel - Blade

- Formularios (ejemplo):
  - En el enrutador definiremos la ruta “mi-ruta” del *action* del formulario y, en el controlador, creamos el método “store()”. Observa cómo recuperamos los datos del formulario a través de la variable *\$r*.

```
Route::post('mi-ruta', 'MiControlador@store');

// Método en MiControlador                                |    // Método alternativo
public function store(Request $r) {                          |    public function store() {
    $email = $r->get("email");                                |        $email = request("email");
    $asunto = $r->get("asunto");                                |        $asunto =
request("asunto");
```

# Laravel - Blade

- Formularios:
  - *Blade* facilita la validación de formularios en el lado del servidor. Se debería hacer también en el lado del cliente (con HTML y JavaScript) para evitar demasiadas peticiones al servidor, mejorar la experiencia de usuario y asegurar que la información enviada es correcta. Pero sobre todo, en un formulario con información sensible.
  - En el cliente es más sencillo: Comprobar que un campo de texto de un formulario no se envía vacío, por ejemplo, es algo trivial con HTML y con Javascript solo un poco más complicado.

# Laravel - Blade

## ■ Formularios:

- En el servidor es más trabajoso: Hay que recibir el formulario, comprobar los valores de los campos y, si hay un error, volver a enviar el formulario respetando los datos que el usuario ya hubiera tecleado junto con un mensaje informando del error.
- También es más seguro. Las validaciones en el cliente pueden desactivarse (usando un navegador antiguo para evitar las comprobaciones por HTML y deshabilitar temporalmente el motor de JavaScript). Las comprobaciones en el servidor son imposibles de saltar para un atacante. Deberíamos hacer los dos conjuntos de comprobaciones en los formularios.

# Laravel - Blade

- Formularios (ejemplo):
  - Vamos a modificar el formulario anterior para que se valide en el servidor:

```
<form method="POST" action="{{ route('mi-ruta') }}">
    @if ($errors->any())
        @foreach ($errors->all() as $error)
            {{ $error }}<br>
        @endforeach
    @endif
    <!-- Resto del formulario igual. -->
</form>
```

# Laravel - Blade

- Formularios (ejemplo):
  - El objeto `$errors` (disponible en todas las vistas) tiene muchos más métodos útiles que puedes consultar en la documentación de Laravel. Y el controlador quedaría así (observa cómo definimos varias reglas de validación para campos del formulario):

```
public function store() {  
    request->validate([  
        'email' => 'required|email',  
        'asunto' => 'required'  
    ]);  
    // A partir de aquí se procesa el formulario igual que  
    antes.  
}
```

# Laravel - Blade

- Formularios (ejemplo):
  - Cuando vuelve a cargarse el formulario que contenía un error, suele ser apropiado mostrarlo con los datos que el usuario ya había tecleado. A esto se le llama “repopular” el formulario, y con *Blade* se hace así (observa el atributo *value* del campo *email*):

```
<form method="POST" action="{{ route('mi-ruta') }}">
    @if ($errors->any()) ...
    @endif
    <input type="email" name="email" value="{{
        old('email') }}" /><br/>
    <!-- Resto del formulario igual. -->
</form>
```

# Laravel - Blade

## ■ CSS y JavaScript:

- Laravel proporciona los archivos “app.css” y “app.js” basados en *Bootstrap* para empezar a trabajar. Para usarlos, basta con añadir a la cabecera de nuestras vistas:

```
<link rel="stylesheet" href="/css/app.css">  
<script src="js/app.js" defer></script>
```

- Si queremos añadir reglas CSS, NO debemos editar “/public/app.css” porque es un CSS compilado y minimizado con SASS.



# Laravel - Blade

## ■ CSS y JavaScript:

- Lo correcto para añadir nuestro CSS a ese archivo sería:
  - Editar “/resources/sass/app.css”.
  - Recompilar este archivo con SASS (o con LESS o stylus).

La recompilación se hace con el comando:

```
$ npm run dev
```

O con:

```
$ yarn dev
```

# Laravel - Blade

- CSS y JavaScript:
  - Guarda tu CSS en el directorio “/public/css” y tu JavaScript en “/public/js”. En ese caso, los archivos no estarían optimizados (tendríamos que optimizarlos a mano si queremos) y serían accesibles de forma pública.

# Laravel - Blade

- Vistas (errores):
  - Un pequeño truco para dar a la aplicación un toque más profesional: Personalizar las vistas de error.
  - Simplemente, crea una carpeta “/resources/views/errors”. Todas las vistas que metas ahí dentro se considerarán pantallas de error.
  - Ahora solo tienes que ponerles los nombres adecuados. Por ejemplo, si creas un archivo llamado “/resources/views/errors/404.blade.php”, esa vista se mostrará cada vez que ocurra un “error 404 (página no encontrada)”.

# Laravel - Blade

## ■ Vistas (errores):

- Si se quiere usar un `@include` de una vista que no existe, Laravel mostraría un error. Si queremos incluir una vista que puede o no estar presente, se debe usar la directiva `@includeIf`:

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

- Si se quiere usar un `@include` de una vista cuando cumpla una determinada expresión booleana que se evalúa como verdadera o falsa, se pueden usar las directivas `@includeWhen` y `@includeUnless`:

```
@includeWhen($boolean, 'view.name', ['status' => 'complete'])  
  
@includeUnless($boolean, 'view.name', ['status' => 'complete'])
```

# Laravel - Blade

- Extras:
  - Secciones interesantes de la documentación de Laravel:
    - **Requests:** Donde recibimos todos los inputs enviados, por ejemplo, desde un formulario.
    - **Sessions:** Para gestionar variables que nos almacenamos en sesión, por ejemplo, de un controlador a otro.
    - **Validation:** Para gestionar las validaciones a efectuar cuando se nos hace una llamada POST y queremos validar el formato de los parámetros recibidos u obligatorios.
    - **API:** Para poder crear APIs con Laravel.
    - **Autenticación:** Para gestionar de una manera rápida y sencilla el registro de usuarios y el login.
    - **Testing:** Para crear y ejecutar los tests de nuestra aplicación.
    - **Roles y permisos:** Permite crear y gestionar los roles y permisos asociados a los usuarios que acceden a la plataforma.

# Laravel - Blade

- Ejercicio resuelto 2: Crear un formulario donde se pidan dos campos, “Email” y “Selector de sectores”. Una vez se haga el *submit* del formulario, mostrar por pantalla los datos seleccionados.

# Laravel - Blade

- Ejercicio resuelto 2 (solución):

*Lo primero es crear dos rutas en el archivo “web.php”, una para pintar el formulario (get) y la otra para recibir el submit (post).*

```
Route::get('/example', 'ExampleController@showForm')->name('example-show');  
Route::post('/example/store', 'ExampleController@storeData')->name('example-store');
```

# Laravel - Blade

- Ejercicio resuelto 2 (solución):  
***Creamos el controlador “ExampleController” con los dos métodos que llamamos desde las rutas.***

```
class ExampleController extends Controller
{
    const SECTORS = array("Electricista", "Albañil", "Fontanero", "Carpintero", "Transportista");

    public function showForm() {
        return view('example-show');
    }

    public function storeData(Request $request) {
        $email = $request->email;
        $sector = $request->sector;

        echo "El email seleccionado es: $email<br>";
        echo "La profesión seleccionada es: <b>" . self::SECTORS[$sector] . "</b>";
    }
}
```



# Laravel - Blade

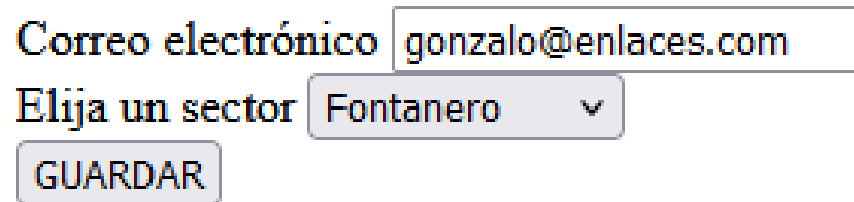
- Ejercicio resuelto 2 (solución):  
***Creamos la vista “example-show.blade.php” donde se muestra el formulario.***

```
<form name="f_prof" id="f_prof" action="{{route('example-store')}}" method="post">
  @csrf
  <div class="form-group">
    <label for="email">Correo electrónico</label>
    <input type="email" class="form-control" id="email" name="email" placeholder="name@example.com">
  </div>
  <div class="form-group">
    <label for="sector">Elija un sector</label>
    <select class="form-control" id="sector" name="sector">
      <option value="0">Electricista</option>
      <option value="1">Albañil</option>
      <option value="2">Fontanero</option>
      <option value="3">Carpintero</option>
      <option value="4">Transportista</option>
    </select>
  </div>
  <button type="submit" class="btn btn-primary">GUARDAR</button>
</form>
```

# Laravel - Blade

- Ejercicio resuelto 2 (solución):

*El formulario se vería así.*



Correo electrónico

Elija un sector

*Y al hacer el “submit”, se imprimiría el resultado seleccionado.*

El email seleccionado es: gonzalo@enlaces.com  
La profesión seleccionada es: **Fontanero**

# Laravel - BD

## ■ Introducción:

- Laravel permite la conexión con las bases de datos MySQL, PostgreSQL, SQLite, SQL Server, MariaDB.
- Además de la configuración vista en el archivo “.env”, también podemos configurar la conexión con la BD en el archivo “/config/database.php”.
- Podemos acceder a más de una BD desde la aplicación. Para ello, cada vez que hagamos una acción contra la BD debemos indicar la conexión a qué BD accedemos:

```
use Illuminate\Support\Facades\DB;  
  
$users = DB::connection('sqlite')->select(/* ... */);
```

- Más en: <https://documentacionlaravel.com/docs/11.x/database>

# Laravel - BD

- Controlador (ejemplo):
  - Podemos hacer diferentes acciones de BD desde un fichero PHP (normalmente un controlador): *select*, *insert*, *update* o *delete*.

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

    return view('user.index', ['users' => $users]);
}
```

# Laravel - BD

- Controlador (ejemplo):
  - La ejecución devuelve un *array* de objetos *stdClass* que se almacena en la variable *\$users*. Por lo que, para acceder a los datos de cada elemento, lo haríamos accediendo como un objeto:

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

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

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

# Laravel - BD

- Controlador (ejemplo):
  - Algunos ejemplos de *insert*, *update* y *delete*:

```
use Illuminate\Support\Facades\DB;  
  
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Marc']);
```

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

```
use Illuminate\Support\Facades\DB;  
  
$deleted = DB::delete('delete from users');
```

# Laravel - BD

- Generador de consultas:
  - Es más común, en lugar de hacer las consultas directas a BD, hacer uso del generador de consultas que incluye Laravel.
  - Proporciona una interfaz más fluida para crear y ejecutar consultas de BBDD.
  - Se puede usar para realizar la mayoría de las operaciones de BD en una aplicación y funciona en todos los sistemas de BBDD compatibles.
  - Utiliza el enlace de parámetros PDO para proteger su aplicación contra ataques de inyección SQL.
  - No es necesario limpiar las cadenas que se pasan como enlaces.

# Laravel - BD

- Generador de consultas (ejemplo):
  - Hacer un *select* contra la tabla *users* como veíamos antes:

```
public function index()
{
    $users = DB::table('users')->get();

    return view('user.index', ['users' => $users]);
}
```



# Laravel - BD

- Generador de consultas (ejemplo):
  - En este caso, la respuesta de la select sería una **colección** “Illuminate\Support\Collection” de objetos *stdClass*. Para recorrer los elementos y obtener su información, podríamos hacer igual que en el caso anterior:

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

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

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

# Laravel - BD

- Generador de consultas (ejemplo):
  - Si en lugar de querer obtener todos los elementos que cumplan un criterio, nos interesara por ejemplo solo el primero, en lugar de utilizar el *get()* haríamos uso de *first()*:

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

En este caso, en lugar de una colección, nos devolvería directamente un objeto *stdClass*.

# Laravel - BD

- Generador de consultas (ejemplo):
  - También podemos hacer ciertas operaciones con una columna al hacer la *select*: *count()*, *max()*, *min()*, *avg()*, etc.:

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

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

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

```
$price = DB::table('orders')
    ->where('finalized', 1)
    ->avg('price');
```

# Laravel - BD

- Generador de consultas (ejemplo):
  - Existen también los métodos *exists()* y *doesn'tExist()* para determinar si nos devolverá algún valor o no:

```
if (DB::table('orders')->where('finalized', 1)->exists()) {  
    // ...  
}  
  
if (DB::table('orders')->where('finalized', 1)->doesn'tExist()) {  
    // ...  
}
```

O también podemos hacer uso del método *distinct()* para evitar filas repetidas:

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

# Laravel - BD

- Generador de consultas (ejemplo):
  - Podemos ejecutar una consulta en *raw* directamente:

```
$users = DB::table('users')
    ->select(DB::raw('count(*) as user_count, status'))
    ->where('status', '<>', 1)
    ->groupBy('status')
    ->get();
```

- Enlazar más de una tabla a través de los *joins*:

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

$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();

$users = DB::table('users')
    ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
    ->get();

$users = DB::table('users')
    ->rightJoin('posts', 'users.id', '=', 'posts.user_id')
    ->get();
```

# Laravel - BD

- Generador de consultas (ejemplo):
  - En el *where*, cuando sea una condición de igualdad, podemos poner el signo =:
- También podrían usarse <, <=, >, >=, != ó *like*:

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

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

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

$users = DB::table('users')
    ->where('name', 'like', 'T%')
    ->get();
```

# Laravel - BD

- Generador de consultas (ejemplo):
  - Podemos anidar todos los *where* que queramos, pero se tendrían que cumplir todos *and*. Si queremos un *OR*, tendremos que usar *orWhere*:

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

- También se puede usar *whereNull*, *whereNotNull*, *whereIn*, *whereNotIn*, etc.

# Laravel - BD

- Generador de consultas (ejemplo):
  - Para los *inserts*, podemos hacer de la siguiente forma.
    - Un registro cada vez:

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

- Varios registros a la vez:

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



# Laravel - BD

- Generador de consultas (ejemplo):
  - Para los *updates*, podemos hacer de la siguiente forma.
    - Se actualiza el registro que cumpla la condición del *where*:

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

- Se actualiza el registro y si no existe, lo crea:

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

# Laravel - BD

- Generador de consultas (ejemplo):
  - El método *upsert* insertará registros que no existen y actualizará los registros que ya existen con nuevos valores que puede especificar.

```
DB::table('flights')->upsert(  
    [  
        ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],  
        ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]  
    ],  
    ['departure', 'destination'],  
    ['price']  
);
```

- El 1º argumento consiste en los valores para insertar o actualizar, mientras que el 2º enumera las columnas que identifican de forma única los registros dentro de la tabla asociada. El 3º es una matriz de columnas que debe actualizarse si ya existe un registro coincidente en la BD.

# Laravel - BD

- Generador de consultas (ejemplo):
  - Para los *deletes*, podemos hacer de la siguiente forma.
    - Borra todos los registros o los que cumplan la condición *where*:

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

- Hace un *truncate* de la tabla (la vacía y reinicia sus autoincrementables):

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

# Laravel - BD

- Migraciones:
  - Constituyen una especie de control de versiones para la BD de la aplicación. Permiten crear y modificar tablas de la BD con independencia del SGBD que estemos usando.
  - Con las migraciones no solo podrás reconstruir la BD, sino que podrás parchear la BD de una aplicación en producción en un tiempo récord y con riesgo cero.

# Laravel - BD

## ■ Migraciones:

- Laravel tiene un sistema de “migraciones” que es como un control de versiones de las modificaciones que se hacen sobre la BD, lo que le permite a un equipo modificar y compartir el esquema de la BD de la aplicación.
- Las migraciones generalmente se combinan con el generador de esquemas de Laravel para construir el esquema de la BD de su aplicación.
- Esto evita agregar manualmente una columna a un esquema de BD local y enviar el código SQL para que otro compañero tenga que hacerlo también y así no le falle la aplicación.

# Laravel - BD

- Migraciones:
  - Dentro de la ruta “/database/migrations” se van generando ficheros cada vez que creamos una nueva migración. Por defecto, al crear el proyecto, ya ha generado algunas:

```
▼ database
  > factories
  ▼ migrations
    🐘 2014_10_12_000000_create_users_table.php
    🐘 2014_10_12_100000_create_password_resets_table.php
    🐘 2019_08_19_000000_create_failed_jobs_table.php
    🐘 2019_12_14_000001_create_personal_access_tokens_table.php
```

# Laravel - BD

## ■ Migraciones:

- Para crear una nueva migración, se debe ejecutar el comando de *Artisan*:

```
php artisan make:migration create_flights_table
```

- Para ejecutar una nueva migración:

```
php artisan migrate
```

- Para forzar la migración:

```
php artisan migrate --force
```

- Para hacer un rollback (deshacer):

```
php artisan migrate:rollback
```

- Para volver a un paso concreto:

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

# Laravel - BD

## ■ Migraciones:

- Para deshacer todas las migraciones:

```
php artisan migrate:reset
```

- Para deshacer todas las migraciones y que se vuelvan a lanzar:

```
php artisan migrate:refresh
```

- Para borrar todas las tablas y lanzar el *migration*:

```
php artisan migrate:fresh
```



# Laravel - BD

- Migraciones:
  - Al crear una migración nueva, genera un nuevo fichero en la ruta de *migrations* con la siguiente estructura:
    - El método *up()* se llamará cuando se ejecute la nueva migración.
    - El método *down()* cuando se haga un *rollback* de la migración.

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

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

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

# Laravel - BD

## ■ Migraciones (ejemplo):

### ○ Acciones con tablas:

- Para crear una nueva tabla *users* con un único campo *id* autoincrementable.

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

Schema::create('users', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->string('email');
    $table->timestamps();
});
```

- Para comprobar si ya existe una tabla o una columna de una tabla.

```
if (Schema::hasTable('users')) {
    // The "users" table exists...
}

if (Schema::hasColumn('users', 'email')) {
    // The "users" table exists and has an "email" column...
}
```

# Laravel - BD

## ■ Migraciones:

### ○ Acciones con tablas:

- Si necesitas modificar una tabla que ya existe (por ejemplo, para añadir o eliminar campos), tienes dos opciones:
  1. Modificar la migración original (en la que se crea la tabla) para añadir o eliminar el campo afectado. Esto te obligará a lanzar la migración de nuevo y, por lo tanto, la tabla se reconstruirá y todos los datos que pudiera contener se perderán.
  2. Crear una nueva migración en la que únicamente se haga la modificación de la tabla, sin tocar el resto. Esto respetará los datos que la tabla ya pudiera contener.

# Laravel - BD

## ■ Migraciones:

### ○ Acciones con tablas:

- Las migraciones pueden usarse para cualquier otra operación sobre la estructura de la BD, como:
  - Cambiar tipos de columnas.
  - Cambiar atributos de columnas (null, unique, default, etc).
  - Cambiar o asignar claves primarias y ajenas.

# Laravel - BD

- Migraciones (ejemplo):
  - Acciones con columnas:
    - Añadir la columna *votes* de tipo *integer* a la tabla *users*.
    - Añadir la columna *email* de tipo *texto* a la tabla *users* que pueda ser *nulable*.

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

Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

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

Schema::table('users', function (Blueprint $table) {
    $table->string('email')->nullable();
});
```

# Laravel - BD

- Migraciones (ejemplo):
  - Acciones con columnas:
    - Modificar una columna *name* a que pueda ser *nullable*.

```
Schema::table('users', function (Blueprint $table) {  
    $table->string('name', 50)->nullable()->change();  
});
```

- Cambiar de nombre una columna de una tabla.

```
Schema::table('users', function (Blueprint $table) {  
    $table->renameColumn('from', 'to');  
});
```

# Laravel - BD

- Migraciones (ejemplo):
  - Acciones con columnas:
    - Borrar una columna de una tabla.

```
Schema::table('users', function (Blueprint $table) {  
    $table->dropColumn('votes');  
});
```

- Borrar varias columnas de una tabla.

```
Schema::table('users', function (Blueprint $table) {  
    $table->dropColumn(['votes', 'avatar', 'location']);  
});
```

# Laravel - BD

- Migraciones (ejemplo):

- Acciones con columnas:

- Crear una clave ajena.

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

Schema::table('posts', function (Blueprint $table) {
    $table->unsignedBigInteger('user_id');

    $table->foreign('user_id')->references('id')->on('users');
});
```

- O más actual así.

```
Schema::table('posts', function (Blueprint $table) {
    $table->foreignId('user_id')->constrained();
});
```



# Laravel - BD

- Migraciones (ejemplo):
  - Acciones con columnas:
    - Crear una clave ajena con opciones de borrado en cascada.

```
$table->foreignId('user_id')  
    ->constrained()  
    ->onUpdate('cascade')  
    ->onDelete('cascade');
```

- Eliminar una clave ajena.

```
$table->dropForeign('posts_user_id_foreign');
```

# Laravel - BD

## ■ Sembrar:

- Laravel incluye un método *seeding* para sembrar su BD con datos de prueba usando clases *seed* (semilla) que se almacenan en el directorio base de “/database/seeder”.
- Las clases semilla pueden tener el nombre que se desee, pero deberían seguir alguna convención como *UsersTableSeeder*, etc.
- De forma predeterminada, se define una clase *DatabaseSeeder* desde la que puede usar el método de llamada para ejecutar otras clases de inicialización, lo que le permite controlar el orden de inicialización.

# Laravel - BD

- Sembrar:

- Para crear un nuevo *seeder* se lanza el comando de *Artisan*:

```
php artisan make:seeder UserSeeder
```

Genera una clase *UserTableSeeder* dentro de la ruta “/database/seeder”. En versiones antiguas de Laravel, esta carpeta se llamaba “seeds”.

# Laravel - BD

- Sembrar:
  - En el método *run()* hacemos los *inserts* correspondientes en la tabla:

```
namespace Database\Seeders;

use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\DB;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Str;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeders.
     *
     * @return void
     */
    public function run()
    {
        DB::table('users')->insert([
            'name' => Str::random(10),
            'email' => Str::random(10).'@gmail.com',
            'password' => Hash::make('password'),
        ]);
    }
}
```

# Laravel - BD

## ■ Sembrar:

- A veces, cuando creamos un nuevo *seeder* no lo detecta y debemos regenerar el *composer auto-loader* lanzando el comando:

```
composer dump-autoload
```

- Para ejecutar el *seeder* podemos utilizar los comandos:

```
php artisan db:seed  
  
php artisan db:seed --class=UserSeeder
```

- El 1º ejecuta todos los *seeders* que están incluidos en el archivo “DatabaseSeeder.php” y en el orden en que se muestran en su método *run()*. El 2º solo ejecuta el método *run()* del *seeder* que especificamos después del parámetro *--class*.

# Laravel - BD

- Sembrar:
  - Si queremos que se ejecuten los *seeders* cuando restablecemos las migraciones, ejecutamos los comandos:

```
php artisan migrate:fresh --seed
```

```
php artisan migrate:fresh --seed --seeder=UserSeeder
```

# Laravel - BD

## ■ Modelos:

- *Eloquent* es una librería incluida con Laravel que utiliza un patrón de software llamado *ORM (Object-Relational Mapping)* para abstraer aún más el acceso a la BD, de manera que no tengamos que escribir y depurar SQL.
- Mapear los objetos de nuestra aplicación con una BD relacional significa que *Eloquent* convierte los registros de tus tablas en objetos de tu aplicación para que los manipules con mayor facilidad. Podrás manejar los datos de tu BD como si fueran objetos de tu aplicación. Y cuando los modifiques, borres o crees, se ejecutará el código SQL necesario para traducir esas operaciones en sentencias para la BD.

# Laravel - BD

## ■ Modelos:

- Laravel supondrá que la tabla se llama igual que el modelo, solo que en minúscula y plural.
- Solo con construir el modelo (y asignarlo a la tabla adecuada) ya tendremos detrás a *Eloquent* haciendo el mapeo objeto-relacional. Luego, ir al controlador y lanzar consultas.
- Más en: <https://documentacionlaravel.com/docs/11.x/eloquent>



# Laravel - BD

- Modelos (ejemplo):

- Para crear un nuevo modelo *Eloquent*:

```
php artisan make:model Flight
```

- Se permite también crear el modelo y una migración de la tabla asociada:

```
php artisan make:model Flight --migration
```

# Laravel - BD

- Modelos (ejemplo):
  - Estos comandos generan una nueva clase PHP en la ruta “app/Models” con el nombre que hemos puesto en el comando, que extiende de la clase *Model*:

```
namespace App\Models;  
  
use Illuminate\Database\Eloquent\Model;  
  
class Flight extends Model  
{  
    //  
}
```

# Laravel - BD

- Modelos (ejemplo):
  - Por convenio, el nombre de la tabla asociada al modelo debe estar en formato “snake case” (convención que compone las palabras separadas por barra baja en vez de espacios y con la primera letra de cada palabra en minúscula). Además, el nombre es en plural (el nombre del modelo es siempre en singular).
  - En el ejemplo, por defecto, la tabla sería *flights*. Si queremos que no cumpla este concepto, se debe especificar en la variable *\$table* de la clase.

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

# Laravel - BD

- Modelos (ejemplo):

- Por convenio, *Eloquent*, establece como clave primaria siempre *id*. Si se quiere establecer otro nombre para la clave primaria, se debe modificar la variable *\$primaryKey* de la clase:

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The primary key associated with the table.
     *
     * @var string
     */
    protected $primaryKey = 'flight_id';
}
```

# Laravel - BD

## ■ Modelos (ejemplo):

- Para obtener la información de la BD basándonos en el modelo:

- Obtendríamos una colección de objetos *Flight* con todos los resultados de la BD.
- También filtrar los datos obtenidos de la BD (solo los activos), ordenados por alguna columna u obtener solo los 10 primeros registros.

```
use App\Models\Flight;

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

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

# Laravel - BD

- Modelos (ejemplo):
  - Otras posibles opciones de obtener un modelo:
    - En vez de obtener una colección de objetos *Flight*, obtendríamos un objeto de tipo *Flight*.

```
use App\Models\Flight;

// 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);
```

# Laravel - BD

- Modelos (ejemplo):
  - También existe la posibilidad de obtener un objeto o crearlo en caso de que no exista ya:

```
use App\Models\Flight;

// Retrieve flight by name or create it if it doesn't exist...
$flight = Flight::firstOrCreate([
    'name' => 'London to Paris'
]);

// Retrieve flight by name or create it with the name, delayed, and
$flight = Flight::firstOrCreate(
    ['name' => 'London to Paris'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);

// Retrieve flight by name or instantiate a new Flight instance...
$flight = Flight::firstOrCreate([
    'name' => 'London to Paris'
]);

// Retrieve flight by name or instantiate with the name, delayed, and
$flight = Flight::firstOrCreate(
    ['name' => 'Tokyo to Sydney'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);
```

# Laravel - BD

- Modelos (ejemplo):

- Para crear un nuevo modelo y hacer un *insert* en la BD:

```
public function store(Request $request)
{
    // Validate the request...

    $flight = new Flight;

    $flight->name = $request->name;

    $flight->save();
}
```

- O también así:

```
use App\Models\Flight;

$flight = Flight::create([
    'name' => 'London to Paris',
]);
```



# Laravel - BD

- Modelos (ejemplo):
  - Para modificar, *update* en BD un modelo:

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

# Laravel - BD

- Modelos (ejemplo):
  - Para averiguar si el modelo o alguna columna del modelo se ha modificado en BD:
    - El método *isDirty* determina si alguno de los atributos del modelo ha cambiado desde que se recuperó el modelo. Se puede pasar un nombre de atributo específico o una matriz de atributos a dicho método para determinar si alguno de los atributos está “sucio”.

```
use App\Models\User;

$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->isDirty(); // true
$user->isDirty('title'); // true
$user->isDirty('first_name'); // false
$user->isDirty(['first_name', 'title']); // true

$user->isClean(); // false
$user->isClean('title'); // false
$user->isClean('first_name'); // true
$user->isClean(['first_name', 'title']); // false

$user->save();

$user->isDirty(); // false
$user->isClean(); // true
```

# Laravel - BD

- Modelos (ejemplo):
  - Para averiguar si el modelo o alguna columna del modelo se ha modificado en BD:
    - El método *isClean* determinará si un atributo no ha cambiado desde que se recuperó el modelo. Este método también acepta un argumento de atributo opcional.

```
use App\Models\User;

$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->isDirty(); // true
$user->isDirty('title'); // true
$user->isDirty('first_name'); // false
$user->isDirty(['first_name', 'title']); // true

$user->isClean(); // false
$user->isClean('title'); // false
$user->isClean('first_name'); // true
$user->isClean(['first_name', 'title']); // false

$user->save();

$user->isDirty(); // false
$user->isClean(); // true
```

# Laravel - BD

- Modelos (ejemplo):
  - Para averiguar si el modelo o alguna columna del modelo se ha modificado en BD:
    - El método *wasChanged* determina si se cambió algún atributo cuando el modelo se guardó por última vez dentro del ciclo de solicitud actual. Se le puede pasar un nombre de atributo para ver si se cambió un atributo en particular.

```
$user = User::create([
    'first_name' => 'Taylor',
    'last_name' => 'Otwell',
    'title' => 'Developer',
]);

$user->title = 'Painter';

$user->save();

$user->wasChanged(); // true
$user->wasChanged('title'); // true
$user->wasChanged(['title', 'slug']); // true
$user->wasChanged('first_name'); // false
$user->wasChanged(['first_name', 'title']); // true
```

# Laravel - BD

- Modelos (ejemplo):
  - Para averiguar si el modelo o alguna columna del modelo se ha modificado en BD:
    - El método *getOriginal* devuelve una matriz con los atributos originales del modelo, independientemente de cualquier cambio en el modelo desde que se recuperó. Se le puede pasar un nombre de atributo específico para obtener el valor original de un atributo en particular.

```
$user = User::find(1);  
  
$user->name; // John  
$user->email; // john@example.com  
  
$user->name = "Jack";  
$user->name; // Jack  
  
$user->getOriginal('name'); // John  
$user->getOriginal(); // Array of original attributes
```

# Laravel - BD

- Modelos (ejemplo):

- Para los *upserts* (vistos antes) podemos hacer así:

```
$flight = Flight::updateOrCreate(  
    ['departure' => 'Oakland', 'destination' => 'San Diego'],  
    ['price' => 99, 'discounted' => 1]  
);
```

- Si existe un vuelo con una ubicación de salida de Oakland y una ubicación de destino de San Diego, se actualizarán sus columnas de precio y descuento.
- Si no existe tal vuelo, se creará un nuevo vuelo que tiene los atributos resultantes de fusionar la primera matriz de argumentos con la segunda matriz de argumentos.

# Laravel - BD

## ■ Modelos (ejemplo):

- Para los *upserts* (vistos antes) también vale así:

```
Flight::upsert([
    ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
    ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
], ['departure', 'destination'], ['price']);
```

- Permite realizar múltiples *upserts* en una sola consulta.
- El 1º argumento del método consiste en los valores para insertar o actualizar. El 2º enumera las columnas que identifican de forma única los registros dentro de la tabla asociada. El 3º es una matriz de las columnas que deben actualizarse si ya existe un registro coincidente en la BD.
- El método *upsert* establecerá automáticamente las marcas de tiempo *created\_at* y *updated\_at* si están habilitadas en el modelo.

# Laravel - BD

- Modelos (ejemplo):
  - Para borrar, *delete* en BD un modelo:

```
use App\Models\Flight;  
  
$flight = Flight::find(1);  
  
$flight->delete();
```

```
Flight::truncate();
```



# Laravel - BD

- Modelos (ejemplo):
  - Una opción interesante es el *SoftDelete*, que permite no borrar la fila de la BD, sino que lo marca como borrado, al añadir una fila *deleted\_at* en la tabla. Añadiríamos en la clase:

```
namespace App\Models;

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

class Flight extends Model
{
    use SoftDeletes;
}
```

# Laravel - BD

- Modelos (ejemplo):
  - Y creamos una migración que nos añadirá la columna *deleted\_at*.

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

Schema::table('flights', function (Blueprint $table) {
    $table->softDeletes();
});

Schema::table('flights', function (Blueprint $table) {
    $table->dropSoftDeletes();
});
```

# Laravel - BD

## ■ Modelos (ejemplo):

- Cuando hacemos consultas a BD, no devuelve aquellos que tengan la columna *deleted\_at* distinto de nulo:

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

- Si queremos obtener las filas “borradas” (marcadas como borradas), ejecutaremos:

- Para restaurarlos en BD.

```
$flight->restore();
```

- Para borrarlos completamente de BD.

```
$flight->forceDelete();
```

# Laravel - BD

- Relaciones entre tablas:
  - Las relaciones se definen como métodos en las clases de modelo Eloquent. Esto proporciona poderosas capacidades de encadenamiento y consulta de métodos. Por ejemplo, podemos encadenar restricciones de consulta adicionales.
    - 1:1 → <https://documentacionlaravel.com/docs/11.x/eloquent-relationships#one-to-one>
    - 1:N → <https://documentacionlaravel.com/docs/11.x/eloquent-relationships#one-to-many>
    - N:M → <https://documentacionlaravel.com/docs/11.x/eloquent-relationships#many-to-many>

# Laravel - Ejemplo CRUD

- Pasos a seguir (los cuatro primeros son solo para aplicaciones nuevas):
  - Instalar y configurar la nueva aplicación.
  - Crear los modelos (se supone que ya tendrás la BD diseñada).
  - Crear las migraciones y los seeders.
  - Lanzar las migraciones y seeders para crear y poblar la BD.
  - Crear en el enrutador las entradas de la funcionalidad que vas a programar.
  - Crear el controlador (si no existe) para la funcionalidad que vas a programar.
  - Crear las funciones del controlador necesarias.
  - Crear las funciones del modelo necesarias (si no existen ya).
  - Crear las vistas necesarias.
  - Probar.
  - Repetir los pasos 5-10 para cada funcionalidad adicional.

# Laravel - Ejemplo CRUD

- Vamos a desarrollar una pequeña **aplicación web desde cero**. Se trata de un fragmento de otra aplicación más grande: una tienda online o tal vez un sistema de gestión de almacén. Nosotros vamos a desarrollar la parte de mantenimiento de productos.
- Para ello, supondremos que en la BD existe una tabla llamada “products” con los campos “id”, “name”, “description” y “Price”.
- Vamos a construir el controlador, el modelo y las vistas necesarias para hacer el CRUD completo de esta tabla con Laravel, sin olvidarnos de las migraciones, los seeders y, por supuesto, el enrutador.

# Laravel - Ejemplo CRUD

## ■ Creación:

- Creamos el proyecto “prueba” desde terminal situándonos en el directorio en cuestión y ejecutando:

```
PS C:\MAMP\htdocs\...>laravel new prueba
```

- En *phpMyAdmin* creamos una BD en blanco con el nombre “prueba” y en el archivo “.env” de nuestro proyecto configuramos la conexión a la misma:

```
DB_CONNECTION=mysql
DB_HOST=localhost
DB_PORT=3306
DB_DATABASE=prueba
DB_USERNAME=root
DB_PASSWORD=root
```



```
$ laravel new prueba

Laravel

Creating a "laravel/laravel" project at "./prueba"
Installing laravel/laravel (v11.4.0)
- Downloading laravel/laravel (v11.4.0)
  - Installing laravel/laravel (v11.4.0): Extracting archive
Created project in C:\MAMP\htdocs\DWES/prueba
```

# Laravel - Ejemplo CRUD

## ■ Migraciones:

- Para esta pequeña aplicación solo necesitamos una migración, puesto que solo tenemos que crear una tabla.
- La migración se crea con el comando:  
`php artisan make:migration create_products_table`  
la cual se escribe en el archivo:  
“/database/migrations/[timestamp]\_create\_products\_table.php”. Al original se le han añadido 3 campos:

```
Schema::create('products', function (Blueprint $table) {  
    $table->id();  
    $table->string('name', 100); // Añadido  
    $table->string('description', 500); // Añadido  
    $table->double('price', 5, 2); // Añadido  
    $table->timestamps();  
});
```



# Laravel - Ejemplo CRUD

## ■ Seeders:

- En este *seeder* vamos a cargar unos cuantos datos de prueba. Obviamente, puedes cambiarlos por los que tú quieras.
- El *seeder* se crea con el comando:

```
php artisan make:seeder ProductTableSeeder
```

que generará el archivo:

“/database/seeder/ProductTableSeeder.php”.

# Laravel - Ejemplo CRUD

## ■ Seeders:

- Recuerda que, para poder lanzar el *seeder* automáticamente con:

```
php artisan migrate:fresh --seed
```

u otro comando similar, tienes que editar el archivo “DatabaseSeeder.php” y añadir al método “run()” la línea:

```
$this->call(ProductTableSeeder::class);
```

- En cualquier caso, siempre podrás lanzar el *seeder* manualmente en cualquier momento con:

```
php artisan db:seed --class=ProductTableSeeder
```

# Laravel - Ejemplo CRUD

## ■ Enrutador:

- El enrutador de la aplicación se encuentra en el archivo “/routes/web.php”. Basta con abrirlo y añadir esta línea:

```
Route::resource('product', 'ProductController');
```

- Alternativamente, podrías crear a mano las siete entradas correspondientes a las siete rutas de un servidor REST. El resultado sería el mismo, pero si defines manualmente las rutas, tienes más control sobre cómo son exactamente.

# Laravel - Ejemplo CRUD

- Enrutador:
  - O podrías hacer algún cambio más profundo a nivel técnico. Por ejemplo, que la petición para hacer “delete” llegue por GET en lugar de por DELETE (así no tendrías que usar un botón de formulario para lanzar el borrado de un producto y podrías lanzarlo con un link).
  - Eso sí, ten en cuenta que, si haces algún cambio de este tipo en las rutas, tu servidor ya no será 100% REST.

# Laravel - Ejemplo CRUD

- Enrutador:

- Hay una posibilidad intermedia: Respetar las 7 rutas estándar REST y añadir alguna adicional que te venga bien, como el borrado mediante GET:

```
// Estas son las 7 rutas estándar REST:
```

```
Route::resource('product', 'ProductController');
```

```
// Añadimos una ruta NO ESTÁNDAR para borrar productos mediante GET:
```

```
Route::get('product/delete/{product}',  
'ProductController@destroy')->name('product.myDestroy');
```

# Laravel - Ejemplo CRUD

- Enrutador:
  - Con la sintaxis utilizada para esta aplicación también hay que modificar el archivo “/bootstrap/app.php” dentro del bloque *withRouting*:

```
use Illuminate\Support\Facades\Route;

return Application::configure(basePath: dirname(__DIR__))
    ->withRouting(
        web: __DIR__.'/../routes/web.php',
        commands: __DIR__.'/../routes/console.php',
        health: '/up',
        then: function () {
            Route::middleware('web')
                ->namespace('App\Http\Controllers')
                ->group(base_path('routes/web.php'));
        }
    )
```

# Laravel - Ejemplo CRUD

- Enrutador (antiguo):
  - Con la sintaxis utilizada para esta aplicación también hay que modificar el archivo “/app/Providers/RouteServiceProvider.php”:

```
Route::prefix('api')
    ->middleware('api')
    ->namespace('App\Http\Controllers') <----- AÑADE ESTO
Route::middleware('web')
    ->namespace($this->namespace)
    ->namespace('App\Http\Controllers') <----- AÑADE ESTO
```

# Laravel - Ejemplo CRUD

## ■ Modelo:

- El modelo se crea con el comando:

```
php artisan make:model Product
```

- El archivo con el modelo se generará en “app/models/product.php”.
- No es necesario que toques este archivo, puedes dejarlo de momento tal y como lo ha generado *Artisan*.



# Laravel - Ejemplo CRUD

## ■ Controlador:

- El controlador de productos se crea con el comando:

```
php artisan make:controller ProductController
```

- El archivo se generará en  
“/app/Http/Controllers/ProductController.php” donde  
tendrás que rellenar el código de los 7 métodos REST  
suministrados en el archivo “ProductController.php”.

# Laravel - Ejemplo CRUD

- Vistas (plantilla principal):
  - La plantilla principal o *master layout* debes crearla en “/resources/views/master.blade.php”.
  - Por supuesto, puedes hacerla como quieras, pero puedes usar de momento el *master layout* visto en el tema sobre *Blade* (proporcionado en el archivo “master.blade.php”) y luego ya lo modificarás a tu gusto.

# Laravel - Ejemplo CRUD

- Vistas (todos los productos):
  - La vista con todos los productos será el suministrado “all.blade.php” y añadirlo en la carpeta “/resources/views”.
- Vistas (creación/modificación de productos):
  - El archivo de la vista será el suministrado “form.blade.php” y añadirlo en la carpeta “/resources/views”.

# Laravel - Ejemplo CRUD

## ■ Funcionamiento:

- Ahora ha llegado el momento de comprobar si tu aplicación funciona. Primero, lanza las migraciones y los *seeders* con:

```
php artisan migrate:fresh --seed
```

- Asegúrate de haber añadido tu *seeder* de productos a “DatabaseSeeder.php” para que se lance automáticamente tras las migraciones. Si todo va bien, la aplicación estará lista para responder en:

*<http://localhost/.../prueba/public/index.php/product>*

# Ejercicios

- Realiza todos los ejercicios del bloque 'Ejercicios 7.2'.

# Laravel - Autenticación

- Introducción:

- La autenticación de usuarios, es decir, el sistema de “login” seguido de la creación de una o varias variables de sesión asociadas al usuario que se acaba de “loguear”, es un componente habitual de las aplicaciones web.
- A partir de Laravel 6, los desarrolladores decidieron aligerar su núcleo lo máximo posible y sacaron del sistema muchos componentes, incluyendo el sistema de autenticación.

# Laravel - Autenticación

## ■ Introducción:

- Actualmente, Laravel proporciona los denominados *Starter Kits*, que son componentes que se pueden instalar mediante *Composer* para realizar esas tareas que se extrajeron del núcleo de Laravel.
- Para la autenticación, Laravel dispone de dos *Starter Kits*, llamados *Breeze* y *Jetstream*. Vamos a ver el primero, que es más sencillo pero suficiente en la mayor parte de las situaciones.
- Más en: <https://documentacionlaravel.com/docs/11.x/authentication>

# Laravel - Autenticación

- Breeze:
  - Contiene todo el código necesario para crear un sistema de autenticación completo y seguro, capaz de:
    - Hacer el “login” e iniciar la sesión.
    - Registrar nuevos usuarios.
    - Recuperar contraseñas olvidadas.
    - Confirmar el registro de usuarios mediante email.



# Laravel - Autenticación

- Breeze:

- Para instalarlo debes abrir un terminal en tu servidor web y ejecutar estos comandos:

```
$ composer require laravel/breeze --dev
$ php artisan breeze:install
Elige "blade"
$ php artisan migrate
$ npm install
$ npm run dev
```

- Los comandos *npm* sirven para compilar el CSS y el JS de *Breeze*. El anterior actualiza las migraciones para crear sus tablas adicionales.

# Laravel - Autenticación

- Breeze:

- Se creará automáticamente varias rutas en un enrutador especial “/routes/auth.php”, entre ellas:

```
Routes::get("/login") // Para mostrar el formulario de login.  
Routes::post("/login") // Para procesar formulario login.  
Routes::post("/logout") // Para cerrar la sesión.  
Routes::get("/register") // Para mostrar formulario registro.  
Routes::post("/register") // Para procesar form. registro.
```

# Laravel - Autenticación

## ■ Breeze:

- También se crearán varios controladores, como “ProfileController”, y resto en “/app/Http/Controllers/Auth”. Y varias vistas, como “login.blade.php”, “register.blade.php” en “/resources/views/auth” y “app.blade.php”.
- Por último, se crea una vista *home* de ejemplo “dashboard.blade.php” a la que llegamos después de hacer “login”. Dicha vista la podrías cambiar en “/app/providers/AppServiceProvider.php” y redirigirla a la que te interese.

# Laravel - Autenticación

- Breeze:

- En las vistas, existen un par de directivas de *Blade* muy útiles relacionadas con las sesiones:

```
@auth
// Este código solo se ejecuta si existe un usuario logueado.
@endauth
```

```
@guest
// Este código solo se ejecuta si NO existe usuario logueado.
@endguest
```

# Laravel - Autenticación

- Clase Auth:

- Además, podemos acceder a los datos del usuario mediante la clase *Auth*:

```
$user = Auth::user() // Devuelve el usuario actualmente  
logueado o null si no hay ninguna sesión abierta.
```

```
if (Auth::check()) { // Devuelve true si el usuario actual  
está logueado.
```

```
...
```

```
}
```

# Laravel - Autenticación

## ■ Middleware auth:

- Hay dos *middlewares* importantes relacionados con la autenticación:

- *Authenticate*, tiene el alias *auth*, que puede usarse en el enrutador.

```
Route::get('/ruta-a-proteger', 'Controlador@metodo')  
->middleware('auth');
```

- *RedirectIfAuthenticated*, alias *guest*.

Nota: Los alias se definían antes en “app/Http/Kernel.php”, ahora en “config/app.php”.

# Laravel - Autenticación

- Middleware auth:
  - También podemos usar estos *middlewares* en el constructor de nuestros controladores para protegerlos en todo o en parte:

```
public function __construct() {  
    // Solo usuarios logueados podrán acceder a  
    cualquier función de este controlador:  
    $this->middleware("auth");  
    // Solo usuarios logueados podrán acceder a los  
    métodos create() y edit():  
    $this->middleware("auth")->only("create", "edit");  
    // Solo usuarios logueados podrán acceder al  
    controlador excepto a show():  
    $this->middleware("auth")->except("show");  
}
```

# Laravel - Autenticación

- Rutas (antiguo):

```
Route::middleware('auth:api')->get('/user', function (Request $request) {  
    return $request->user();  
});
```

- Aquí estaría la ruta que se creaba en el “api.php” asociada (“/user”) y con el *middleware* de *auth:api* asignado.



# Laravel - Sesiones

- Introducción:
  - Laravel también proporciona su propio sistema de manejo de variables de sesión, es decir, variables persistentes en el servidor asociadas a cada cliente.
  - Las variables de sesión de Laravel son mucho más seguras y poderosas que las variables de sesión estándar de PHP.
  - Las sesiones se configuran en “/config/sessions.php”. Existen varios drivers de sesión: *files* (driver por defecto), *memcached* y *redis* (para aplicaciones en producción) y *database* (para seguridad adicional).

# Laravel - Sesiones

- Introducción:
  - Laravel también proporciona su propio sistema de manejo de variables de sesión, es decir, variables persistentes en el servidor asociadas a cada cliente.
  - Las variables de sesión de Laravel son mucho más seguras y poderosas que las variables de sesión estándar de PHP.
  - Las sesiones se configuran en “/config/sessions.php”. Existen varios drivers de sesión: *files* (driver por defecto), *memcached* y *redis* (para aplicaciones en producción) y *database* (para seguridad adicional).

# Laravel - Sesiones

- Variables *flash*:
  - Son variables de sesión que solo duran una petición y luego se autodestruyen. Se usan típicamente para enviar un “feedback” o mensaje de retroalimentación al usuario.
  - Ejemplo: Imagina el típico formulario de “login”. En caso de producirse un error, lo habitual es que la aplicación nos muestre de nuevo ese formulario con un mensaje del tipo “Usuario no reconocido”.

# Laravel - Sesiones

## ■ Variables *flash*:

- Para eso, haríamos lo siguiente en el controlador. Observa el uso del método “with()” para crear una variable *flash* de sesión llamada *mensaje*.

```
return ('login/form')->with('mensaje', 'Usuario no  
reconocido');
```

- En la vista, podemos acceder a esa variable *flash*.

```
@if (session('mensaje'))  
    {{ session('mensaje'); }}  
@endif // A partir de este momento, la variable flash  
se destruye y cualquier intento de acceder a  
ella provocará un error de ejecución.
```

# Laravel - Sesiones

- Variables convencionales:
  - Se manejan con la clase *Session*, que tiene métodos estáticos para crear variables, destruirlas, consultarlas, etc. Los métodos más útiles son:
    - `put()`: Almacena una variable de sesión.

```
Session::put('nombre-variable', 'valor');
```

- `push()`: Elimina una variable de sesión.

```
Session::push('nombre-variable');
```

# Laravel - Sesiones

## ■ Variables convencionales:

- `get()`: Devuelve el valor de una variable de sesión.

```
$v = Session::get('nombre-variable'[, 'valor-por-defecto']);
```

- `all()`: Devuelve todas las variables de sesión en un array.

```
$a = Session::all('nombre-variable', 'valor');
```

- `flush()`: Elimina todas las variables de sesión.

```
$a = Session::flush();
```

- `flash()`: Crea manualmente una variable de sesión tipo flash.

```
$a = Session::flash('nombre-variable', 'valor');
```

# Laravel - Helpers

## ■ Introducción:

- Un *helper* es un componente del framework diseñado para facilitar alguna tarea típica en el desarrollo de una aplicación web.
- Los helpers cambian mucho de una versión a otra de Laravel, por lo que es recomendable echar un vistazo a la documentación oficial para saber cuales están disponibles en tu versión de Laravel.
- Más en: <https://documentacionlaravel.com/docs/11.x/helpers>

# Laravel - Helpers

- Ejemplo:

- El helper *asset* sirve para generar una ruta absoluta a partir de una relativa, de modo que la ruta absoluta siempre funcione, sea cual sea el servidor donde ejecute la aplicación:

```
<a href="{{ asset('/users/login') }}">Volver</a>
```

Cuando actúe, traducirá la expresión anterior por el código HTML:

```
<a href='https://miservidor.com/users/login'>Volver</a>
```



# Laravel - Helpers

- Otro ejemplo:

- El helper *route* sirve para rutas con nombre. Si en el enrutador tenemos una ruta con un nombre como éste:

```
Route::get("mi-ruta", "mi-controlador@metodo") -  
>name("nombre-ruta");
```

Podemos referirnos a ella en cualquier vista como:

```
<a href="{{asset('mi-ruta')}}">Texto</a><!--1ª forma-->  
<a href="{{route('nombre-ruta')}}">Texto</a><!--  
2ª forma-->
```

La 2ª forma es mejor que la 1ª porque permite cambiar en cualquier momento la dirección que ve el usuario sin modificar el código fuente del resto de la aplicación.

# Laravel - Helpers

- Más ejemplos:
  - request <https://documentacionlaravel.com/docs/11.x/helpers#method-request>
  - redirect <https://documentacionlaravel.com/docs/11.x/helpers#method-redirect>

# Laravel - Localización

## ■ Introducción:

- Las características de localización de Laravel ofrecen una forma conveniente de recuperar cadenas en varios idiomas, lo que te permite soportar fácilmente múltiples idiomas dentro de tu aplicación.
- Si deseas personalizar los archivos de idioma de Laravel, puedes publicarlos a través del comando:

```
php artisan lang:publish
```

- Más en: <https://documentacionlaravel.com/docs/11.x/localization>

# Ejercicios

- Realiza el proyecto 'Sitio web con Laravel'.