# RECURSOS

[Documentación oficial PHP]

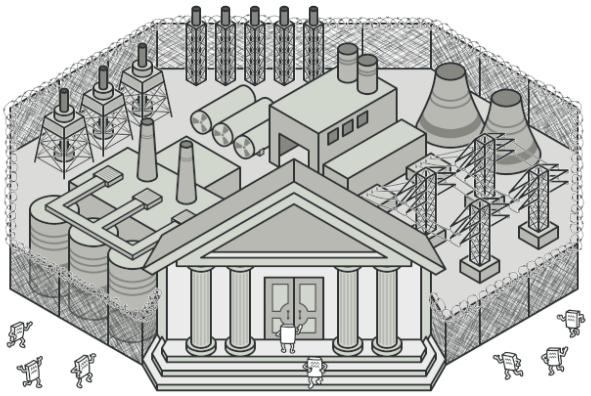
[Documentación de Laravel]

[Laravel API]

**NAMESPACES EN PHP**

Cuando empezamos a trabajar con Laravel, una de las primeras cosas que os pueden llamar la atención son los **espacios de nombres** o **namespaces**.

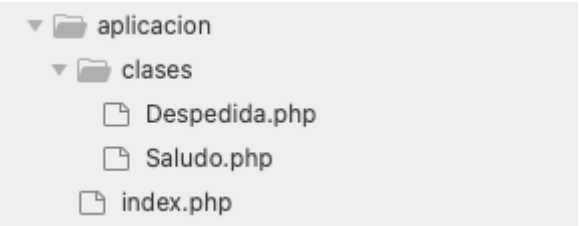
Laravel utiliza un patrón de diseño que se conoce como **FACADE** (Fachada) cuyo objetivo reside en estructurar el código y reducir su complejidad. Esto se consigue con una división en bloques que conlleva la minimización de dependencias entre clases. Esto es, a través de una *fachada*, se nos proporciona una interfaz simple a un subsistema complejo que contiene muchas partes móviles.



Todos estos elementos se organizan en diferentes carpetas dentro del framework. Y aquí es donde hacen acto de aparición los **namespaces**. Según la documentación de PHP, los espacios de nombres se utilizan para encapsular elementos, que precisamente es lo que se persigue el patrón de *diseño facade*. Podríamos decir que un espacio de nombres es como un ***contenedor***.

Cuando contamos con una gran cantidad de clases e interfaces, es posible que algunas entren en conflicto, por lo que es necesario organizarlas en diferentes espacios de nombres para evitar este tipo de problemas.

Supongamos la estructura de carpetas de la imagen:



Definimos la clase **`Saludo`**.

```
namespace Aplicacion\Clases ;

class Saludo
{
      public static function saludar()
      {
            echo "¡Hola gente!" ;
      }
}
```

La primera línea nos garantiza que la clase **`Saludo`** existe para el espacio de nombres indicado. Aunque se puede utilizar cualquier nombre, el estándar **PSR-4** (PHP Standard Recommendation) recomienda utilizar la ruta, desde la raíz de nuestro proyecto hasta donde se encuentra definido el elemento, para definir el **`namespace`**. Ahora necesitamos utilizar esta clase en **`index.php`**. ¿Cómo lo hacemos?

```
require_once "clases/Saludo.php" ;
Saludo::saludar() ;
```

Si abrimos ahora la aplicación en el navegador obtendremos un **`Fatal error: Uncaught Error: Class 'Saludo' not found`**. Esto sucede, porque la clase se encuentra en un espacio de nombres y, si queremos hacer uso de ella, tendremos que indicarle a PHP que necesitamos tener acceso a los elementos de dicho contenedor.

```
require_once "clases/Saludo.php" ;
use Aplicacion\Clases\Saludo ;
Saludo::saludar() ;
```

Aunque esta es la forma más usual y recomendada, también podemos hacerlo de otra manera:

```
require_once "clases/Saludo.php" ;
Aplicacion\Clases\Saludo::saludar() ;
```

Supongamos ahora que en la carpeta **`clases`** tenemos definida la clase **`Despedida`**. Si quisiéramos utilizarla desde **`index.php`**, haríamos igual que hemos hecho con Saludo; esto es:

```
require_once "clases/Saludo.php" ;
require_once "clases/Despedida.php" ;
```

```php
use Aplicacion\Clases\Saludo ;
use Aplicacion\Clases\Despedida ;

Saludo::saludar() ;
Despedida::despedir() ;
```

Sin embargo, desde PHP10 podemos simplificar lo anterior de la siguiente manera:

```php
use Aplicacion\Clases\{Saludo, Despedida} ;
```

Aunque no es muy común en Laravel, además de todo lo que hemos visto, PHP nos permite asignar alias a cada módulo utilizando la palabra reservada **`as`**.

```php
use Aplicacion\Clase\Despedida as Adios ;
```

Entonces, a partir de ahora no podremos hacer lo siguiente:

```php
Despedida::despedir() ;
```

Sino que tendremos que utilizar el alias.

```php
Adios::despedir() ;
```

# LARAVEL

## INTRODUCCIÓN

**LARAVEL** es un framework PHP que nos proporciona numerosas herramientas para el desarrollo de aplicaciones web y servicios. Trabajar con este framework requiere de un servidor web, un motor de bases de datos y un intérprete PHP. Utilizamos para ello un paquete como **XAMPP** - distribución gratuíta de Apache que contiene MariaDB, PHP y Perl - o **DOCKER**, que ha sido la herramienta que hemos utilizado a lo largo del curso para configurar nuestros servicios.

Como vamos a utilizar Docker, necesitaremos montar el gestor de dependencias **COMPOSER** en el contenedor donde estamos corriendo el servicio web, para lo cual basta con seguir las instrucciones de instalación que encontramos en su página web.

El ecosistema de **LARAVEL** nos permite trabajar con multitud de herramientas que, por falta de tiempo, desgraciadamente quedan fuera de este curso. No obstante, mencionamos a continuación algunas de ellas:

**LARAVEL BREEZE**
Kit que incorpora características de autenticación básicas (login, registro, recuperación de contraseñas, verificación de emails y confirmación de contraseña). Trabajaremos con él durante el trimestre.

**LARAVEL FORGE**
Herramienta diseñada para facilitar al máximo el aprovisionamiento, la administración del servidor, las tareas de mantenimiento y el monitoreo, independientemente del marco utilizado para desarrollar una aplicación.

**LARAVEL HERD**
Herramienta únicamente para MacOS que facilita el desarrollo local de aplicaciones web con Laravel.

**LARAVEL INERTIA**
Herramienta que facilita el desarrollo de aplicaciones SPA (Single Page Application) usando Laravel permitiendo que una aplicación del lado del servidor renderice los resultados en el cliente.

**LARAVEL JETSTREAM**
Kit de inicio que incorpora funcionalidades de autenticación y control de sesiones, verificación de correo, autenticación 2F, soporte API a través de Laravel Sanctum, etc.

**LARAVEL SANCTUM**
Añade una capa de autenticación basada en API utilizada con proyectos SPA y/o aplicaciones móviles, facilitando las tareas de autenticación de forma segura a través de tokens de sesión.

**LARAVEL LIVEWIRE**
Framework del ecosistema Laravel que permite el desarrollo de componentes sin necesidad de utilizar lenguaje del lado del cliente. De esta manera, se establece una comunicación fluída entre cliente y servidor sin necesidad de recargar la página.

**LARAVEL PINT**
Herramienta que facilita el mantenimiento del código y corrige el estilo manteniendo su consistencia.

**LARAVEL SAIL**
Herramienta que permite dockerizar el entorno de desarrollo Laravel a través de una interfaz de comandos sencilla que facilitará la interacción con el entorno de desarrollo en Docker sin necesidad de tener experiencia en Docker.

**LARAVEL SOCIALITE**
Paquete que nos permite añadir a nuestras aplicaciones web la posibilidad de iniciar sesión en aplicaciones de terceros, como Facebook, X, Google, Github, Bitbucket, Steam y LinkedIn.

# EMPEZANDO CON LARAVEL

## LARAVEL SAIL

A partir de esta su última versión, el framework incorpora **Laravel Sail**, una sencilla herramienta que nos permitirá interactuar fácilmente con **Docker** a través de la terminal del sistema, y sin necesidad de tener experiencia con esta última. **Sail** proporciona un entorno para el desarrollo de aplicaciones Laravel utilizando PHP y un motor de bases de datos como **MySQL** o **MariaDB**.

### LARAVEL SAIL EN SISTEMAS WINDOWS

Trabajar con Windows, Docker y Laravel Sail puede llegar a convertirse en un auténtico dolor de cabeza. Antes de nada, recuerda que tendrás que tener habilitada en la BIOS el sistema de virtualización, ya que vamos a trabajar con el **Subsistema de Windows para Linux** o **WSL** (Windows Subsystem for Linux) que nos permitirá ejecutar un entorno Linux en la máquina Windows, sin necesidad de una máquina virtual independiente ni de arranque dual.

Abre la **Power Shell** y comprueba que está configurada la versión **wsl2** del Subsistema de

Windows para Linux.

```
wsl -l -v
```

Si no tienes configurada la **versión 2** tendrás que hacerlo con el siguiente comando:

```
wsl --set-default-version 2
```

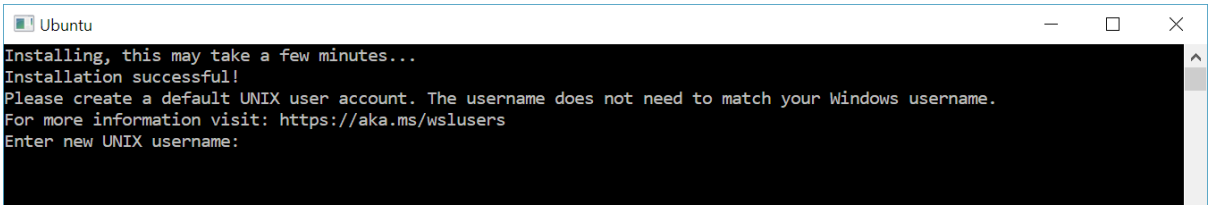
Hecho esto, puedes proceder con la instalación:

```
wsl --install
```

Generalmente se instalará por defecto la última versión de Ubuntu. Sin embargo, podemos elegir la distribución Linux que deseemos, bien a través de la Microsoft Store o con el siguiente comando:

```
wsl --distribution <nombre_de_la_distribución>
```

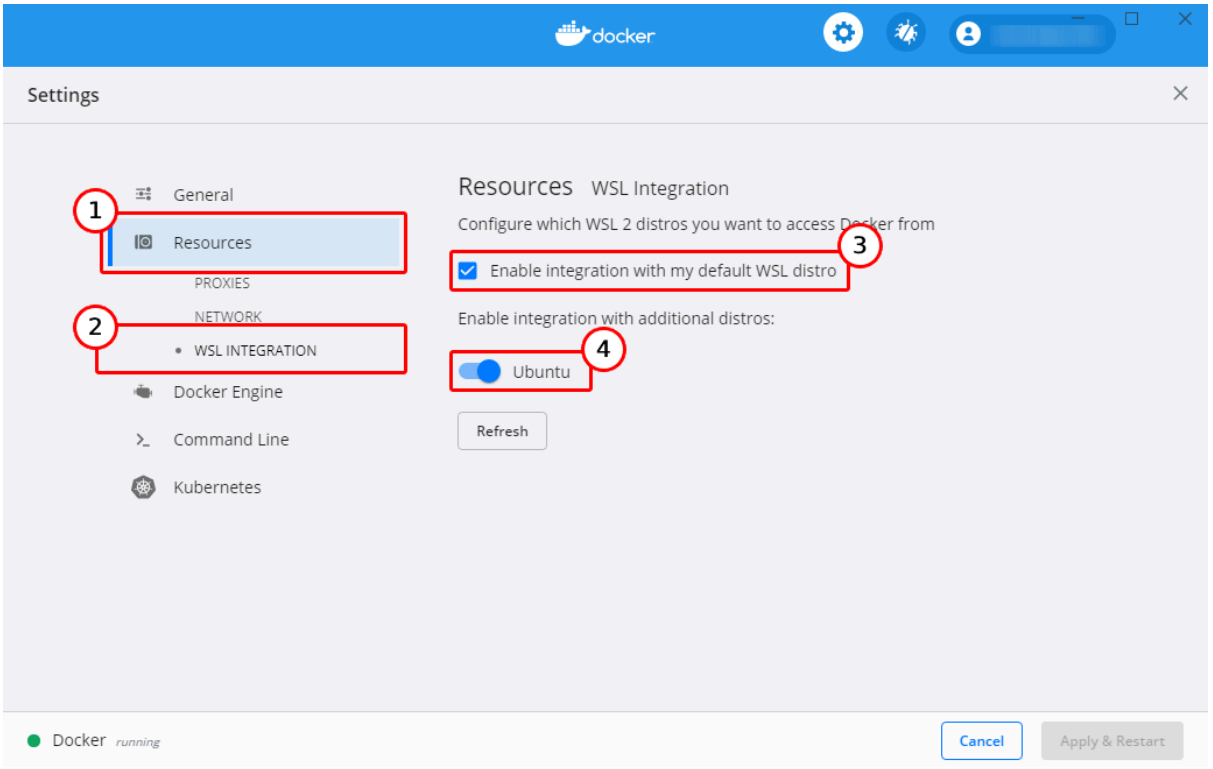
Finalizada la instalación, se nos mostrará un mensaje similar al que vemos en la siguiente imagen y se nos pedirá que creemos una cuenta de usuario y contraseña.




Hecho esto, accedemos a la distribución Linux y, seguidamente, procedemos a realizar las comprobaciones que vimos en el **anterior apartado**. Además, tendremos que instalar la versión de PHP que necesitemos, así como las extensiones que mencionamos, junto con la herramienta PHP-FPM.

Mientras completamos las instalaciones necesarias dentro del Subsistema Windows para Linux, configuramos Docker para que integre la distro instalada con **WSL2**.

Hacemos esto accediendo al menú de configuración y seguidamente en **Resources / WSL Integration**. En éste último menú marcamos la opción **Enable integration with my default WSL Distro** y las distribuciones instaladas.




### Instalar dependencias en Linux

Si trabajamos en un entorno Linux bastará bastará con tener instalado en nuestro equipo **Docker**, **Composer**, **Node.js** (los necesitaremos para trabajar con Laravel aunque no hagamos uso de Laravel Sail) y la última versión de PHP.

Generalmente, los sistemas Linux suelen instalar una versión de PHP por defecto, así que tendremos que comprobar si es la que necesitamos e instalarla si es una versión diferente. Recuerda que puedes comprobar la versión de PHP lanzando el siguiente comando desde la terminal:

```
php --version
```

Tendremos que comprobar si tenemos instaladas determinadas extensiones de PHP que serán necesarias para trabajar con Laravel. Ejecuta el siguiente comando y comprueba si están instaladas las extensiones **`bz2`**, **`curl`**, **`mbstring`**, **`intl`**, **`mysql`**, **`zip`** y **`xml`**.

```
php -m
```

Si no las tienes instaladas, tendrás que hacerlo ejecutando los siguientes comandos.

```
# Añadimos el repositorio necesario
sudo add-apt-repository ppa:ondrej/php

# Actualizamos la lista de paquetes
sudo apt update
```

*Estos apuntes fueron elaborados por Antonio Sánchez Bujaldón, modificados por Alicia Vega Moreno y se comparten aquí con su autorización y créditos correspondientes.*

```
# Instalamos las extensiones necesarias
sudo apt install php8.2
sudo apt install php8.2-cli php8.2-bz2 php8.2-curl php8.2-mbstring php8.2-intl
sudo apt install php8.2-xml php8.2-zip
```

Se recomienda instalar igualmente la herramienta **PHP-FPM** (FastCGI Process Manager) que permite a PHP procesar peticiones web de forma más rápida y eficiente.

```
# Instalamos php-fpm
sudo apt install php8.2-fpm

# Habilitamos la herramienta en el archivo de configuración apache
sudo a2enconf php8.2-fpm
```

Antes de empezar a trabajar con `sail` se recomienda revisar que el parámetro DB_HOST en el archivo **`.env`** tenga el valor **`mysql`** (nombre del servicio creado por Laravel Sail para la conexión con el motor de bases de datos). En caso de que no esté configurado correctamente, tendríamos que cambiar el valor de dicho parámetro por **`mysql`**.

## NUESTRO PRIMER PROYECTO

Abrimos la terminal del sistema, nos movemos a nuestra carpeta de trabajo e iniciamos el proyecto Laravel utilizando **Composer** lanzando el siguiente comando:

```
composer create-project --prefer-dist laravel/laravel <nombre_proyecto>
```

La opción **`--prefer-dist`** instalará la última versión del framework. Sin embargo, podemos elegir la versión que deseemos.

```
composer create-project laravel/laravel <nombre_proyecto> "5.8"
```

Es más, podemos elegir la subversión más actualizada de una versión.

```
composer create-project --prefer-dist laravel/laravel <nombre_proyecto> "5.8.*"
```

El comando anterior genera un *scaffolding* (andamio) con la estructura de directorios y archivos con los que vamos a trabajar para crear nuestro proyecto. Generado el andamiaje del proyecto necesitaremos instalar las dependencias y librerías, para lo que bastará con lanzar los siguientes comandos:

```
composer install
npm install
```

cd ap

## TRABAJANDO CON LARAVEL SAIL

Terminado de configurar nuestro sistema operativo, y creado nuestro proyecto Laravel siguiendo los pasos indicados más arriba, accedemos a la carpeta del proyecto e instalamos Laravel Sail utilizando **Artisan**.

```
php artisan sail:install
```

El proceso de instalación nos preguntará qué tipo de base de datos vamos a utilizar para nuestro proyecto, así que seleccionamos **mysql** o **mariadb**. **Laravel Sail** generará el archivo **`docker-compose.yml`** con los servicios de **Docker**, a partir de la configuración elegida.

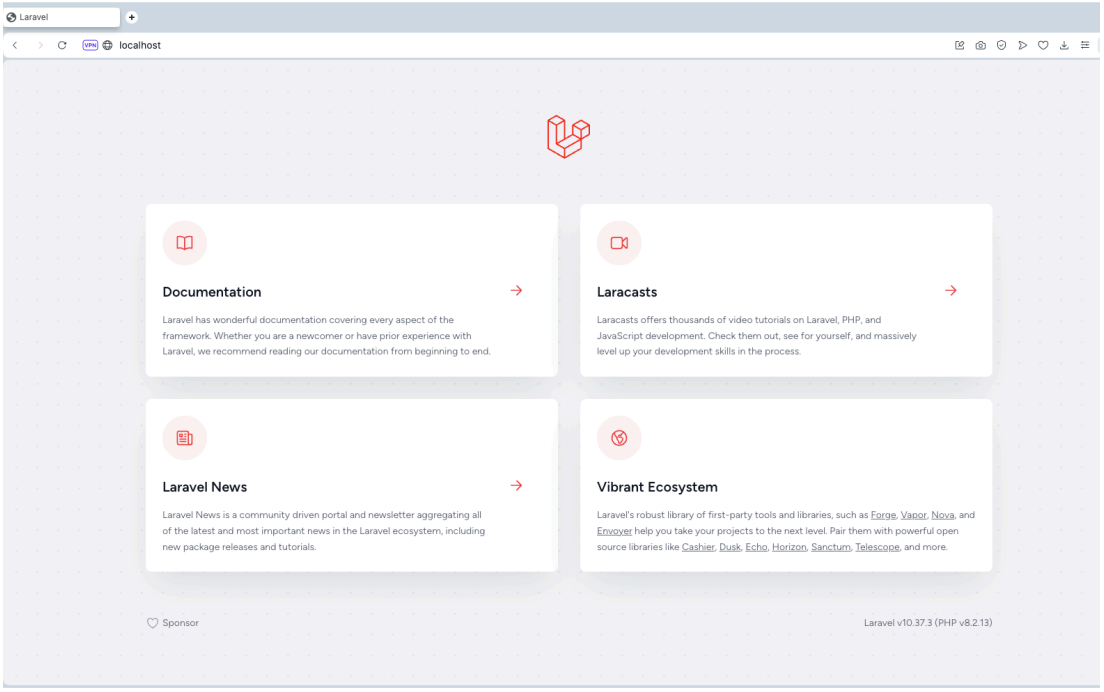
Las dependencias de **Laravel Sail** se descargan automáticamente cuando creamos un proyecto Laravel. Si ésto no es así o no estamos seguros de que se hayan descargado, siempre podemos volver a hacerlo con el siguiente comando:

```
composer require laravel/sail --dev
```

Tras finalizar el proceso de instalación, y tras asegurarnos de tener abierto **Docker**, lanzamos nuestra herramienta con el siguiente comando.

```
./vendor/bin/sail up    # comando en Linux|MacOs
```

Recuerda que, si estás trabajando en Windows, el comando anterior se deberá ejecutar desde la distribución Linux que hayamos instalado. Seguidamente, escribimos **`localhost`** en la barra de dirección del navegador, y accederemos a la web que por defecto genera el instalador de Laravel.




*Estos apuntes fueron elaborados por Antonio Sánchez Bujaldón, modificados por Alicia Vega Moreno y se comparten aquí con su autorización y créditos correspondientes.*

Detenemos el servicio.

```
./vendor/bin/sail stop
```

A partir de ahora, cualquier comando tendremos que lanzarlo a través del comando **`sail`**. Esto es, si queremos conocer los comandos de **Artisan** haremos:

```
./vendor/bin/sail php artisan list
```

Si vamos a trabajar con **Laravel Sail** será mejor que creemos un alias, ya que utilizaremos el comando cada vez que queramos realizar cualquier acción con Laravel.

```
alias sail="./vendor/bin/sail"
```

Puedes obtener una lista de los comandos que puedes utilizar con **Laravel Sail** llamando directamente al comando `sail`. Los comandos más habituales son:

```
sail up                 # levanta los contenedores de Docker
sail stop               # detiene los contenedores de Docker
sail down               # detiene y elimina los contenedores de Docker

sail artisan <comando> # ejecuta un comando Artisan dentro del contenedor
sail tinker             # inicia la consola de Laravel en el contenedor
sail shell              # inicia una shell dentro del contenedor
sail mysql              # conecta con el servidor de base de datos que se está
                        # ejecutando en el contenedor Docker
```

## ESTRUCTURA DEL PROYECTO

Cuando creamos un proyecto de Laravel se generará automáticamente la estructura básica o scaffolding, que contará con los siguientes directorios y archivos con los que vamos a trabajar para crear nuestro proyecto.

a. **archivo `.ENV`**
   Archivo de configuración rápida. No es necesario hacer uso de él, se recomienda utilizarlo para no tener que recurrir a los archivos propios de configuración de Laravel.

b. **carpeta `BOOTSTRAP`**
   El contenido de esta carpeta no tiene nada que ver con el framework CSS Bootstrap. Contiene los archivos necesarios para inicializar y lanzar la aplicación web, además de la carpeta **`CACHE`** utilizada por el framework para optimizar el rendimiento. **NO TOCAR**.

c. **carpeta `CONFIG`**
   Contiene los archivos de configuración. Se recomienda tener mucho cuidado si no se sabe lo que se hace.

d. **carpeta `DATABASE`**
Contiene las subcarpetas **`FACTORIES`** (factorías), **`MIGRATIONS`** (migraciones) y **`SEEDS`** (semilleros); opcionalmente también pueden contener el directorio **`SCHEMA`** con el script SQL de creación de la base de datos.

e. **carpeta `PUBLIC`**
Debe contener los archivos que son accesibles para la aplicación: **imágenes**, **favicon**, **robots**, etc. Además, contiene el archivo **`index.php`** que se supone una especie de **controlador frontal**.

f. **carpeta `RESOURCES`**
Contiene la carpeta **`VIEWS`** donde guardaremos las vistas, así como el código **`CSS`** y **`JS`** previos a su compilación. En versiones anteriores, esta carpeta albergaba también por defecto los archivos de localización. A partir de Laravel 10 tendremos que hacer:

```
php artisan lang:publish
```

El comando anterior genera la carpeta **`LANG`** con los archivos de idiomas en la raíz del proyecto.

g. **carpeta `ROUTES`**
En su interior encontramos los archivos de rutas. En este curso, en la medida de lo posible, nos centraremos únicamente en **`web.php`** y **`api.php`**.

h. **carpeta `STORAGE`**

Contiene tres directorios: **`APP`**, **`FRAMEWORK`** y **`LOGS`**. Nosotros nos centramos únicamente en la primera que almacenará los archivos generados por nuestra aplicación, como por ejemplo, los cargados desde el cliente.

i. **carpeta `VENDOR`**
Guarda las dependencias de composer. **IMPORTANTE**: **NO TOCAR**. Generamos esta carpeta con el comando: composer install.

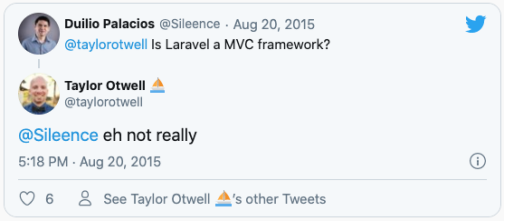
j. **carpeta `APP`**
Contiene el corazón de la aplicación y, aunque dentro de ella podemos encontrar numerosos elementos, nosotros nos centraremos en unos pocos.

- **carpeta `EXCEPTIONS`**: contiene el manejador de excepciones. Guardaremos en el directorio cualquier clase que creemos para gestionar las excepciones de nuestra aplicación.

- **carpeta `HTTP`**: encontramos los **controladores** y los ***middlewares*** (más adelante veremos en qué consisten).

- **carpeta `MODELS`**: contiene los **modelos** que definamos.

Según su desarrollador, Taylor Otwell, Laravel no sigue exactamente un patrón Modelo-Vista-Controlador. Como veremos más adelante, aunque utilicemos los constructores básicos de dicho patrón de diseño, Laravel necesita de capas adicionales para poder organizar los aspectos básicos de cualquier aplicación. Es más, en versiones anteriores no existía una carpeta Modelos al uso.




Sin embargo, el clamor popular ha obligado a que, a partir de **Laravel 9**, se incluya dicha carpeta en el *scaffolding* encontrándose en la ruta **app** → **models**.

- **carpeta PROVIDERS**: los proveedores, como veremos más adelante, extienden las funcionalidades básicas de la aplicación, permitiendo por ejemplo *bindear* servicios, registrar eventos o realizar cualquier otra tarea sobre una petición HTTP entrante.
- **carpeta RULES**: nos servirá para guardar reglas de validación. Esta carpeta no aparece por defecto, sino que se creará automáticamente cuando creemos nuestra primera regla de validación.

Existen más carpetas (**BROADCASTING**, **CONSOLE**, **EVENTS**, **JOBS**, **LISTENERS**, **MAILS** y **POLICIES**) que podremos añadir al directorio APP y que, llegado el momento podrían cubrir determinadas necesidades de la aplicación.

## MIGRACIONES

Creamos la base de datos utilizando **migraciones**, que nos permitirán, no sólo construir la estructura de la base de datos de nuestra aplicación, sino también exportarla cómodamente y llevar el control de versiones:

```
php artisan make:migration create_<nombre_tabla>_table
```

Realmente, una **migración** no es más que un objeto PHP que define la estructura de una tabla de la base de datos. Será necesario seguir la regla de nombrado que se ha utilizado en el ejemplo anterior a la hora de crear la migración.

Hemos de tener mucho cuidado con el orden de creación de las migraciones. Esto es debido a que, **Artisan** ejecutará las migraciones en el orden en que las hemos creado y, por tanto, las tablas de la base de datos se crearán en el mismo orden. De esta manera, no podemos crear en primer lugar una tabla que dependa de otra.

Cada migración contiene dos métodos: **`up`** y **`down`**. El primero de ellos, se encargará de ejecutar la migración y, por tanto, de crear la tabla de la base de datos. El método **`down`** se ejecutará siempre que hagamos un **rollback** (se aconseja eliminar la tabla).

```
Schema::create('usuario', function (Blueprint $table)
{
    $table->id('idUsu') ;  // crea un campo autoincremental (clave primaria)
    $table->string('nombre');
    $table->string('apellido') ;
    $table->string('email')->unique() ;
    $table->string('pass') ;
    $table->timestamps() ;
}) ;
```

Observa que hemos utilizado diferentes métodos asociados al objeto **`Blueprint`** que define la estructura de la tabla que estamos creando.

| Método | Descripción |
|---|---|
| **`id(atributo)`** | Crea un campo autoincremental de tipo **`BIGINTEGER`** y lo configura como **clave primaria**. El nombre por defecto para el campo es **`id`**. |
| **`string(atributo, longitud)`** | Crea un campo de tipo **`VARCHAR`** que admite cadenas de la longitud indicada (parámetro opcional). |
| **`unique()`** | Define un índice en la tabla indicando que el valor del atributo debe ser único. |
| **`timestamps()`** | Crea en la tabla los campos **`created_at`** y **`updated_at`** de tipo **`TIMESTAMP`**, que serán gestionados automáticamente por Laravel siempre que se inserte o actualice un registro. |

Observa dos detalles: hemos cambiado el nombre de la tabla de **`users`** a **`usuario`** y, además, hemos definido un nombre para la clave primaria. Estos dos cambios tendrán sus consecuencias ya que para **Eloquent** el nombre de las tablas es siempre en plural y los campos de clave primaria se llaman **`id`**. Esto implica que Eloquent no funcionará correctamente, por lo que tendremos que hacer las modificaciones pertinentes para reconducir su funcionamiento, que son las siguientes:

1. Debemos cambiar el **nombre del modelo** correspondiente (archivo y clase).

2. Cambiamos el nombre de la migración sustituyendo **`users`** por **`usuario`**.
3. Cambiamos el nombre de la clase por **`CreateUsuarioTable`**.
4. Cambiamos el nombre de la tabla en los métodos **`up`** y **`down`** de la migración.
5. Si hemos generado el *scaffolding* de autenticación tendremos que modificar, tal y como vemos a continuación, el archivo **`auth.php`** contenido en la carpeta **`config`**.

```
'providers' => [
        'users' => [
            'driver' => 'eloquent',
            'model' => App\Models\Usuario::class,
        ],
```

Aún resta una modificación más por hacer, pero la veremos más adelante cuando estudiemos los **modelos**.

Supongamos ahora que vamos a crear la migración para la tabla **`TAREA`**. Haremos:

```
php artisan make:migration create_tarea_table
```

Artisan creará el esqueleto de la migración que nosotros completamos como sigue:

```
public function up(): void
{
    Schema::create('tarea', function (Blueprint $table) {
        $table->id('idTar') ;
        $table->unsignedBigInteger('idUsu') ;

        $table->text('texto') ;
        $table->boolean('completa')->default(false) ;
        $table->date('fecha') ;

        $table->foreign('idUsu')->references('idUsu')->on('usuario')
              ->onDelete('cascade') ;
    });
}
```

Al crear esta tabla hemos definido la clave foránea existente entre **`USUARIO`** y **`TAREA`**. Observa que hemos encadenado varios métodos:

1. **`foreign`** nos permite definir el campo de clave foránea en **`TAREA`**.
2. Con **`references`** indicamos con qué atributo de la tabla **`USUARIO`** vinculamos el anterior.
3. Finalmente, el método **`on`** especifica en qué tabla está la clave referenciada.

A modo de ejercicio, termina de construir la base de datos con las dos tablas que restan: **`ETIQUETA`** y la resultante de la relación **`N:N`** que llamaremos **`TAREA_ETIQUETA`**.

```php
// migración TAREA
Schema::create('tarea', function (Blueprint $table) {
    $table->id('idTar');
    $table->date('fecha') ;
    $table->text('texto') ;
    $table->boolean('completa')->default(false) ;
});

// migración TAREA_ETIQUETA
Schema::create('tarea_etiqueta', function (Blueprint $table) {
    $table->id('idTE') ;
    $table->unsignedBigInteger('idTar') ;
    $table->unsignedBigInteger('idEti') ;

    $table->foreign('idTar')->references('idTar')->on('tarea')
          ->onDelete('cascade') ;
    $table->foreign('idEti')->references('idEti')->on('etiqueta')
          ->onDelete('cascade') ;
});
```

Creadas las migraciones necesarias, tendremos que lanzarlas contra la base de datos, que deberíamos haber creado previamente a través de la terminal o de PHPMyAdmin; a no ser que estemos trabajando con **Docker**, que habrá creado automáticamente el servicio y la base de datos, que se llamará como hayamos definido en el archivo de configuración **`.ENV`**.

Utilizaremos a continuación el comando:

```
php artisan migrate
```

Cada migración definida se examinará y las ejecutará convenientemente, creando las tablas en la base de datos. Si deseamos realizar un rollback tendremos que utilizar el siguiente comando:

```
php artisan migrate:rollback
```

Si, por otro lado, queremos refrescar las migraciones haremos:

```
php artisan migrate:refresh
```

## MODIFICADORES

Hemos visto anteriormente algunos ejemplos de modificadores. Resumimos en la siguiente tabla algunos de los más habituales, aunque podrás encontrar más información al respecto en la documentación oficial de Laravel.

| Modificador | Significado |
|---|---|
| **`autoIncrement()`** | Define la columna como clave primaria autoincremental (**`INTEGER`**). |

| | |
|---|---|
| `default(valor)` | Define un valor por defecto para la columna. |
| `invisible()` | Hace que la columna sea invisible para cualquier consulta `SELECT *` |
| `nullable()` | Permite valores **nulos** para la columna. |
| `unsigned()` | Hace que el valor de una columna `INTEGER` pueda tomar únicamente valores sin signo. |
| `primary(atributo)` | Define el atributo como clave primaria. |
| `primary([att1, …])` | Define una clave primaria compuesta. |
| `unique(atributo)` | Define el valor del atributo como único. |

### CLAVES FORÁNEAS

Aunque anteriormente hemos visto cómo crear una clave foránea, puntualizamos en este apartado algunas cuestiones a tener en cuenta.

```
Schema::table("nombre_tabla", function(Blueprint $table) {
    ...
    $table->foreign("clave_foranea")
          ->references("clave_primaria")
          ->on("tabla") ;
}) ;
```

Utilizando este método debemos de tener en cuenta que debemos definir un atributo de clave foránea **obligatoriamente** de tipo `UNSIGNED BIG INT`. Sin embargo, podemos ahorrarnos este paso y simplificar la creación de la clave foránea como sigue:

```
Schema::table("nombre_tabla", function(Blueprint $table) {
    ...
    $table->foreignId("clave_foranea")
          ->constrained() ;
}) ;
```

En este caso, será Laravel quien cree automáticamente el atributo de clave foránea. Sea como sea, podremos añadir también acciones por defecto.

```
$table->foreignId("clave_foranea")
      ->constrained()
      ->onUpdate("cascade")
      ->onDelete("cascade")
```

## SEMILLEROS

Los semilleros o *seeders* nos proporcionan un mecanismo para rellenar de datos las tablas de la base de datos. Creamos un semillero de datos con el comando:

```
php artisan make:seeder <nombre_tabla>Seeder
```

A diferencia de lo que ocurría con las migraciones, no es necesario seguir ningún criterio para el nombrado, sin embargo es aconsejable seguir el que acabamos de indicar. Los *seeders* se crearán en la carpeta del mismo nombre y contendrán el método **`run`** que incluirá la lógica necesaria para insertar en la tabla los datos que deseemos.

```
class MiModeloTablaSeeder extends Seeder
{
       public function run()
       {
           DB::table("...")->insert([ "campo" => valor, ... ]) ;
       }
}
```

Realizamos las operaciones de inserción de datos en la tabla en el método **`run`** y haremos uso de los métodos que pone a nuestra disposición la clase **`DB`** que, previamente deberemos importar y estudiaremos más adelante.

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

Seguidamente lanzamos el siguiente comando para ejecutar los semilleros.

```
./vendor/vin/sail db:seed
```

## FACTORÍAS

Utilizaremos los seeders cuando queramos poblar alguna tabla con datos prefijados. Sin embargo, las **factorías** nos ayudarán a realizar el poblado de la base de datos con datos aleatorios. Invocamos las factorías desde la clase **`DatabaseSeeder`** aunque previamente tendremos que haber definido el **modelo** correspondiente a la tabla.

Poblar la base de datos con los *seeders* es bastante útil cuando queremos introducir unos pocos registros con información definida *a priori*. Sin embargo, los *seeders*, resultan especialmente incómodos si queremos poblar la base de datos con bastantes registros con información generada de forma aleatoria.

¿Cómo creamos las factorías? En primer lugar tendremos que crear el **modelo** de cada una de las tablas que deseemos poblar, ya que usaremos la factoría a través de este. Sea como sea, podemos crear la factoría directamente.

```
php artisan make:factory <nombre_modelo>Factory
```

Se creará una clase como la que vemos a continuación:

```
use App\Models\MiModelo ;
```

```php
use Illuminate\Database\Eloquent\Factories\Factory;

class MiModeloFactory extends Factory
{
    protected $model = MiModelo::class;

    public function definition()
    {
        return
        [
            ...,
        ];
    }
}
```

Seguidamente, en el *array* devuelto por la sentencia **`return`** especificamos, de forma asociativa, el nombre del atributo de la tabla y el valor que deseamos asignarle. Una de las ventajas que presentan las factorías es que ya nos proporcionan un objeto de clase **`Faker`** a través del cual podemos generar valores aleatorios. Supongamos que queremos poblar la tabla de usuarios.

```php
return [
  "nombre"   => $this->faker->name(),
  "email"    => $this->faker->email,
  "password" => Hash::make("12345678"),
] ;
```

Hecho esto, invocamos la factoría a través del modelo correspondiente (en este caso **`Usuario`**), y lo haremos desde la clase **`DatabaseSeeder`** que, como sabemos, será la encargada de invocar a los *seeders* y/o factorías que poblarán nuestra base de datos. En este caso, haremos lo que sigue dentro del método **`run`** de dicha clase.

```php
\App\Models\MiModelo::factory(10)->create() ;
\App\Models\MiModelo::factory()->times(10)->create() ;
```

Utilizando cualquiera de las líneas anteriores, se insertarán **10 registros** con datos aleatorios.

Aunque no es una norma y podemos utilizarlas indistintamente, se aconseja utilizar las **Factorías** siempre que queramos poblar nuestras tablas con datos aleatorios. Sin embargo, emplearemos **Seeders** cuando la información esté previamente definida. Obviamente, podemos combinar ambas técnicas ya que, desde un *seeder* también es posible invocar una factoría, tal y como hemos hecho anteriormente.

**IMPORTANTE**: si creamos la factoría de esta manera, no debemos olvidarnos de **importar el modelo** y asignar la clase correspondiente a la propiedad **`$model`**. Sin embargo, esto no

será necesario, si nos basamos inicialmente en el modelo para generar la factoría, para lo cual crearemos la factoría como sigue.

```
php artisan make:factory <nombre_modelo>Factory --model=nombre_modelo
```

Pero, ¿qué es un modelo?

## MODELOS

Como sabemos, Laravel incorpora un **ORM** llamado **Eloquent**, que mapea los datos almacenados en la base de datos sobre objetos PHP y viceversa. De esta manera, escribiremos un código portable e independiente, no siendo necesario (en la mayoría de las ocasiones) manipular la base de datos directamente con SQL.

Cada entidad de la base de datos se corresponderá con un modelo, que nos permitirá interactuar con dicha tabla, permitiéndonos hacer consultas, actualizaciones, borrados, etc., de manera cómoda y sencilla. Creamos un modelo en Laravel a través de ARTISAN.

```
php artisan make:model <nombre_del_modelo>
```

El modelo se creará automáticamente en la carpeta **`App/Models`**. Laravel, impone ciertas restricciones a los modelos, para que nos sea más fácil trabajar con ellos.

a. Según la documentación de Laravel, el nombre del modelo se escribe en singular, en contraposición al de la tabla en la base de datos. Sin embargo, por convención, nosotros siempre nombramos las tablas en singular. ¿Cómo solucionamos esto? Fácil: bastará con indicarle a Eloquent el nombre real de nuestra tabla, sobrecargando la propiedad protegida **`$table`**.

```
protected $table = "nombre_real_de_la_tabla" ;
```

b. El nombre de las claves primarias, por defecto, es **`id`**. No obstante, si utilizamos una notación diferente, bastará con indicarlo a Eloquent a través del siguiente atributo.

```
protected $primaryKey = "nombre_de_la_clave_primaria" ;
```

Existen otras propiedades interesantes que conviene reseñar. Antes de esto, necesitamos conocer el concepto de **asignación en grupo** (**Mass assignment**) que hace referencia a un mecanismo de protección intrínseco a los modelos de Laravel, impidiendo que un tercero pueda modificar los valores del modelo.

Supongamos, por ejemplo, que una vez definido el usuario, queremos que su dirección de email no pueda modificarse, pero sí su nombre, apellidos, dirección, etc. Cuando presentemos al usuario el formulario de edición de su perfil es posible que, si tiene conocimientos básicos de informática, pueda modificar la dirección de email alterando el campo a través del **inspector de elementos** del navegador. Aunque podemos evitar esto

(y lo haremos) haciendo uso del sistema de validación de Laravel, podemos combinar esta protección con el uso de las propiedades **`fillable`** y **`guarded`** de los modelos de Eloquent.

| Propiedad | Visibilidad | Significado |
|---|---|---|
| **`fillable`** | `protected` | *Array* donde añadimos los atributos del modelo que pueden ser asignables en masa. |
| **`guarded`** | `protected` | Inverso al anterior. *Array* donde se especifican los atributos del modelo que no pueden ser asignables de forma masiva. |
| **`hidden`** | `protected` | *Array* donde se añaden los atributos del modelo que quedarán ocultos cuando se haga una consulta en la base de datos. |
| **`casts`** | `protected` | *Array* que permite asociar aquellos atributos del modelo que deseemos al tipo nativo en la base de datos. |
| **`timestamps`** | `public` | Nos permite indicar mediante un valor de verdad (`true/false`) si hemos definido en el modelo los campos **`created_at`** y **`updated_at`**. |

## MÉTODOS DE ELOQUENT

**Eloquent** nos proporciona un conjunto de métodos que nos ayudarán a realizar fácilmente operaciones (**consultas**, **borrado**, **inserción** y **actualización**) sobre cada uno de los modelos de nuestra aplicación. Debemos recordar que, si Eloquent nos devuelve un objeto de tipo **`Builder`** y/o **`Collection`** será necesario utilizar **`get`** o **`first`** para recuperar los registros. Sin embargo, si el tipo devuelto es **`Collection`** también podemos emplear **`all`**.

### SELECCIÓN

A continuación se muestran algunos de los métodos de Eloquent que emplearemos más frecuentemente para realizar consultas sobre los modelos.

```
Modelo::all()                   # devuelve todos los registros de la tabla
Modelo::find(id)                # devuelve el registro con id indicado
Modelo::find([id, id, ...])     # devuelve los registros con los id indicados.
Modelo::findOrFail(id)          # devuelve el registro con id o lanza la
                                # excepción ModelNotFoundException

Modelo::orderBy('campo','asc|desc')
```

También podemos realizar búsquedas en la base de datos utilizando la cláusula **`WHERE`** de SQL.

```
Modelo::where('campo', 'op_relacional', valor)
```

Lo anterior ejecuta una consulta como:

```
SELECT * FROM Modelo WHERE campo op_relacional valor ;
```

Si el operador es de igualdad, podemos omitir el segundo parámetro. Eloquent permite utilizar también funciones de agregación.

```
# devuelve el número de registros
Modelo::count()
Modelo::where(...)->count()

# devuelve el valor máximo y mínimo de una columna
Modelo::max('campo')
Modelo::min('campo')

# devuelve la media de los valores de una columna
Modelo::avg('campo')
```

## INSERCIÓN

La manera más sencilla de insertar un modelo nuevo en la base de datos consiste en crear el modelo, asignar un valor a sus propiedades y, finalmente llamar a su método **`save`.**

```
$modelo = new Modelo ;
$modelo->propiedad = valor ;
...
$modelo->save() ;
```

Sin embargo, no es precisamente el más cómodo. En su lugar, podemos emplear **`create`.**

```
Modelo::create([ clave1' => valor,
                 clave2' => valor,
                 ...])
```

Como el método **`create`** es considerado por Laravel como un método de **asignación en masa**, deberás especificar previamente - si es necesario según la aplicación que estés desarrollando - en el modelo qué campos de la tabla pueden ser insertados a través del atributo **`$fillable`**.

```
protected $fillable = ['campo1', 'campo2', ...] ;
```

Otro método que nos puede resultar de utilidad es **`firstOrCreate`** encargado de buscar en la base de datos el registro indicado y devolverlo. Si dicho registro no existe, lo crea.

```
Modelo::firstOrCreate([...]) ;
```

También podemos escribirlo como sigue:

```
Modelo::firstOrCreate([ 'campo1' => valor, ...],
                      [ 'campoN' => valor, ...]) ;
```

En este caso, el método buscará un registro cuyos atributos coincidan con los valores expresados en el primer *array* asociativo. Si lo encuentra, lo devuelve; en otro caso, lo crea con los valores indicados para todos los campos, tanto en un *array* como en otro. Sería un equivalente a:

```
INSERT IGNORE INTO modelo (campo1, campo2, ..., campoN)
VALUES (..., ... );
WHERE campo1 = valor, campo2 = valor, … ;
```

De forma similar podemos hacer uso del siguiente método:

```
Modelo::updateOrCreate([ 'campo1' => valor, ...],
                       [ 'campoN' => valor, ...]) ;
```

Este método buscará los registros que coincidan con las parejas clave valor indicadas en el primer *array* y, si los encuentra, los actualiza con los valores expresados en el segundo *array*; en otro caso, inserta el registro y lo devuelve. Recuerda que estos dos últimos métodos son de **asignación masiva**. Consideramos este método de inserción y/o de…

### ACTUALIZACIÓN

Podemos actualizar los modelos de dos formas diferentes. En primer lugar, buscamos el registro que queremos modificar, cambiamos los campos deseados y, por último, utilizamos **`save`** para actualizar.

```
$modelo = Modelo::find(...)
$modelo->propiedad = valor
...
$modelo->save()
```

O bien, ejecutamos el siguiente comando de Eloquent:

```
Modelo::where(...)->update([ 'clave' => valor, ... ])
```

### BORRADO

La manera más natural de realizar un borrado consiste en buscar el modelo, guardarlo en una variable y, a continuación utilizar el método **`delete`** para eliminarlo.

```
$modelo = Modelo::find(...)
$modelo->delete()
```

También podemos utilizar los siguientes métodos:

```
Modelo::destroy(id, id, ...)
Modelo::where(...)->delete()
Modelo::find(...)->delete()
```

### BORRADO «SUAVE»

Si utilizamos la técnica de **borrado suave**, los registros no se borran realmente de la base de datos, sino que se añade información a un campo llamado **`deleted_at`** relativa a la **fecha** en que se ha producido el borrado. Así, si este atributo tiene un valor diferente de **`NULL`**, Eloquent interpretará que el registro ha sido borrado, aunque siga existiendo en la base de datos.

Debes recordar que, para poder hacer uso de esta técnica, deberás añadir el campo **`deleted_at`** a la hora de crear la tabla. Además, tendremos que indicarle al modelo que queremos emplear el **borrado suave**. Importamos en primer lugar el **trait** correspondiente.

```
use Illuminate\Database\Eloquent\SoftDeletes ;
```

Y lo usamos dentro del modelo:

```
use SoftDeletes;
```

Hecho esto, si borramos un modelo y seguidamente recuperamos todos sus registros, obtendremos un listado de todos **los que no se han eliminado**.

```
Modelo::all() ;
```

Sin embargo, en la base de datos aún existen dichos registros, sólo que se han marcado temporalmente como eliminados. Básicamente es como si los hubiésemos tirado a una papelera de reciclaje. Si quisiéramos obtener información de todos los registros, borrados o no, haremos lo siguiente:

```
Modelo::withTrashed()->get()
Modelo::withTrashed()->where(...)->get()
```

Si sólo queremos visualizar o buscar entre los registros contenidos en la papelera de reciclaje.

```
Modelo::onlyTrashed()->get()
Modelo::onlyTrashed()->where(...)->get()
```

Estos registros eliminados temporalmente pueden ser recuperados. Realmente, cuando hacemos esto, el campo deleted_at se marca a **`NULL`** en la base de datos para el registro elegido.

```
Modelo::onlyTrashed()->find(...)->restore()
```

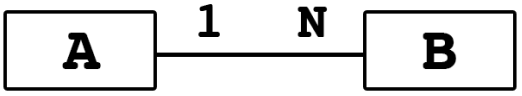
Si, por el contrario, queremos eliminar definitivamente un registro de la papelera de reciclaje, haremos:

```
Modelo::onlyTrashed()->find(...)->forceDelete()
```

## RELACIONES

### UNO A MUCHOS

Definimos también en el modelo las relaciones existentes entre objetos. Supongamos la siguiente relación:




A la hora de desarrollar nuestra aplicación tendremos dos modelos: **A** y **B**, y lógicamente tendremos que establecer la relación entre ambos para que Eloquent asocie los elementos de una y otra tabla.

De esta manera, mirando la relación desde **A** hacia **B**, nos encontramos con una relación **uno a muchos**. Utilizamos el método **`hasMany`**, definido en la clase **`Model`** de la que hereda nuestro modelo, para recuperar los modelos de tipo **B** asociados al modelo **A**.

En este sentido, crearemos un método en el modelo **A** que nos devuelva todos los modelos de **B** con los que está relacionado **A**.

```
public function unoamuchos()
{
    return $this->hasMany('App\Models\B','A_primary_key') ;
}
```

Observa que lo hacemos de esta manera porque el atributo de clave foránea se llama igual que la clave primaria en **A**. Si esto no sucede, y ambos atributos se llaman de forma diferente, tendremos que especificarlo como sigue:

```
public function unoamuchos()
{
    return $this->hasMany('App\Models\B','B_foreign_key', 'A_primary_key') ;
}
```

Laravel nos permite simplificar lo anterior de la siguiente manera:

```php
public function unoamuchos()
{
    return $this->hasMany(B::class, ...) ;
}
```

El método **hasMany** devuelve un objeto de clase **HasMany**, por lo que podremos recurrir a los métodos que esta clase nos proporciona para gestionar los resultados. Si queremos acceder al conjunto de modelos **B** utilizaremos los métodos **getResults**, **get** y/o **first**. Suponiendo que **$a** es un modelo de clase **A**.

```php
# Devuelve una colección de modelos de clase B
# SELECT * FROM ...
$a->unoamuchos()->getResults() ;

# Devuelve una colección de modelos de clase B
# SELECT * FROM ...
# SELECT columna, ... FROM
$a->unoamuchos()->get(['columna', ...]) ;

// Devuelve un modelo (el primero) de clase B
$a->unoamuchos()->first() ;
```

Es **importante** recordar que, con los dos primeros métodos se obtiene un objeto de tipo **Collection**, en tanto que con el último tenemos un objeto de **clase B**.

### MUCHOS A UNO

Si miramos la relación anterior de derecha a izquierda, tenemos una relación **muchos a uno**. De forma análoga a como hiciéramos anteriormente, creamos en el modelo **B** un método que devuelva el modelo **A** con el que está relacionado el primero. En este caso utilizamos el método **belongsTo**.

```php
public function muchosauno()
{
      return $this->belongsTo(A::class, 'B_foreign_key') ;
}
```

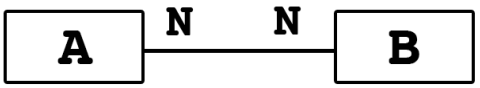
Observa que tenemos que especificar el nombre del atributo de clave foránea debido a que no seguimos las convenciones de Laravel en el nombrado de claves primarias. En este texto hemos optado por llamar a la clave foránea de un modelo igual que la clave primaria de la tabla con que se relaciona. Sin embargo, si optamos por llamarlos de forma diferente, tendremos que indicarlo.

```php
public function muchosauno()
{
      return $this->belongsTo(A::class, 'B_foreign_key', 'A_primary_key') ;
}
```

El método **belongsTo** devuelve un objeto de clase **BelongsTo**, cuyos métodos, igual que los de **hasMany**, podrás encontrar en la documentación de Laravel.

### MUCHOS A MUCHOS

A la hora de trabajar con relaciones muchos a muchos, nos enfrentamos a tres tablas: **A**, **B** y una tabla intermedia que contiene, como claves foráneas, las claves primarias de **A** y **B**. La conjunción de estas claves definen la clave primaria de la tabla intermedia que, en este caso llamaremos **AB**.




En tal situación, crearemos únicamente los modelos correspondientes a las entidades **A** y **B**. Entonces, ¿cómo accedemos a la tabla intermedia? **Eloquent** lo hará automáticamente por nosotros, para lo cual tendremos que crear, en primer lugar, y en los dos modelos, un método que nos devuelva los registros con los que existe un vínculo.

Por ejemplo, en el modelo **A** creamos un método que, para un determinado registro nos devuelva una colección de elementos de **B** con los que está relacionado.

```
public function modelosB()
{
  $this->belongsToMany('App\Models\B', 'A_B', 'idA', 'idB') ;
}
```

En esta ocasión, utilizamos el método **belongsToMany** donde cada parámetro representa:

a. Modelo con el que establecemos la relación. Podemos escribirlo como en el ejemplo anterior, o bien indicando el nombre de la clase **B::class**.
b. Nombre de la tabla intermedia o **pivote**.
c. Clave primaria de la tabla|modelo **A** (donde estoy).
d. Clave primaria de la tabla|modelo **B** (con quien me relaciono).

Como veremos más adelante, **Eloquent** nos proporciona métodos para añadir, borrar y actualizar registros fácilmente. ¿Cómo? A través de **A** o de **B** y haciendo uso de los siguientes métodos:

```
$modeloA->modelosB()->attach(idB, ['campo1' => valor, ...])
```

Observamos que, opcionalmente, como segundo parámetro podemos pasar un *array* asociativo especificando el valor para los conocidos como **elementos pivotes**, esto es, los atributos que puedan existir en la relación.

Si, por el contrario, queremos eliminar el vínculo entre los modelos **A** y **B** haremos lo siguiente:

```
$modeloA->modelosB()->detach(idB)
```

La actualización de un registro de la relación se llevará a cabo con el siguiente método, que también nos permite pasar un *array* asociativo con los posibles nuevos valores para los atributos de la relación:

```
$a->modelosB()->updateExistingPivot(idB, [ 'campo1' => valor, ... ])
```

El método **`sync`** es otro de los más utilizados cuando trabajamos con relaciones **muchos a muchos**. Generalmente, utilizamos este método para sincronizar la relación entre dos modelos, asegurándose de que, por ejemplo, un modelo **A** esté asociado exactamente con los modelos de **B** con el **`id`** especificado.

```
$a->modelosB()->sync([id, id, ...])
```

Existen variantes de este método que, además de lo anterior, nos permite restablecer vínculos o actualizar registros en la tabla intermedia. En último lugar, el siguiente método nos permite mostrar información de los atributos existentes en la relación intermedia.

```
$a->modelosB()->withPivot('campo_de_la_relación')
```

Esto daría como resultado un objeto similar al siguiente:

```
App\Models\ModeloB {#4359
    idB: ...,
    ...
    pivot: Illuminate\Database\Eloquent\Relations\Pivot {#3421
        idA: ...,
        idB: ...,
        ...
    }
}
```

# CONTROLADORES

Como es habitual, debemos recurrir al comando **`artisan`** para crear un nuevo controlador.

```
php artisan make:controller NombreController
```

Automáticamente se creará el controlador en la carpeta **`App/Http/Controller`**. Cada controlador será una clase que extiende de **`Controller`** y, dentro de dicha clase crearemos los métodos que responderán a las diferentes solicitudes.

```php
class TestController extends Controller
{
    public function foo(Request $req) { ... }
}
```

Cada uno de estos métodos podrá recibir o no parámetros. Generalmente, recibiremos un objeto de tipo **`Request`** que encapsula información sobre la petición **HTTP** actual. Si mostramos el contenido del objeto **`$request`** obtendremos información que, en ocasiones puede resultarnos de muchísima utilidad.


A través de dicho objeto, además podemos tener acceso, de forma transparente, a los parámetros **GET**|**POST** adjuntos en la solicitud.

```php
$req->has("parametro")
$req->input("parametro" [, valor_por_defecto])
```

El método **`has`** devuelve **`true`**|**`false`**, según se haya recibido o no el parámetro indicado; en tanto que, **`input`** devolverá el valor, al cual podremos indicarle un valor por defecto, que nos será devuelto en caso de que el parámetro especificado no exista.

A continuación se citan algunos de los métodos que más utilizaremos de la clase **`Request`**.

| Método | Significado |
|---|---|
| `all()` | Devuelve un *array* con todos los datos recibidos en la petición. |
| `query()` | Devuelve un *array* con todos los valores de la **query string** (los que aparecen en la URL tras el símbolo **?**) |
| `isMethod(metodo)` | Comprueba si el protocolo HTTP utilizado en la petición es el indicado como parámetro. |
| `url()` | Devuelve la URL sin la **query string**. |

| | |
|---|---|
| `fullUrl()` | Devuelve la URL completa. |

# REQUESTS

Para poder trabajar en el controlador con los parámetros de las llamadas de la API nos vamos a crear una Request de la siguiente manera:

```
php artisan make:request NombreRequest
```

# DEPURACIÓN EN LARAVEL

Si queremos visualizar información sobre alguna variable podemos utilizar alguna de las siguientes funciones/helpers: `dd` y/o `dump`. La única diferencia que hay entre ambas es que la primera detiene la ejecución del proceso.

# RUTAS EN LARAVEL

El sistema de **routing** de Laravel se encarga de manejar el flujo de solicitudes y respuestas, desde y hacia el cliente. Cada ruta se define a través de su URL y el método HTTP utilizado. Laravel pone a nuestra disposición diferentes archivos de rutas, entre los que utilizaremos nosotros: **`web.php`** y **`api.php`**. Será en el primero de ellos en el que definamos las rutas para nuestra aplicación web, en tanto que en el segundo crearemos las correspondientes a una posible **API**. De momento, nos centraremos en **`web.php`**.

Tal y como se comentó en temas anteriores, HTTP define diferentes verbos que serán utilizados para realizar/atender solicitudes HTTP. A través de la clase **`Route`** y en función del verbo que deseemos utilizar, definir una ruta es relativamente sencillo.

## GET

Se utiliza siempre que queramos **recuperar** información.

```
Route::get("url", ...) ;
```

## POST

Se utiliza, además de cuando **enviamos** información sensible, cuando queremos **guardar** información (insertar un registro).

```
Route::post("url", ...) ;
```

## PUT

Se utiliza cuando queremos **actualizar** un registro completo en la base de datos. Sólo cuando estamos desarrollando una API.

```
Route::put("url", ...) ;
```

## DELETE

Se emplea cuando queremos **borrar** información. Sólo cuando estamos desarrollando una API.

```
Route::delete("url", ...) ;
```

## PATCH

Se emplea cuando queremos **modificar parcialmente** la información de un registro de la base de datos. Sólo cuando estamos desarrollando una API.

```
Route::patch("url", ...) ;
```

## PARÁMETROS EN LAS RUTAS

Generalmente, como sabemos, la definición de una ruta puede incorporar como segundo parámetro una función anónima. Aunque ya veremos que esto puede cambiar. Esta función puede recibir un objeto de tipo **`Request`**, que ya estudiamos anteriormente, que contiene información sobre la petición realizada por el cliente.

```
Route::get("url", function(Request $request) { ... }) ;
```

Aunque no es necesario en un controlador, emplear este parámetro en el script de rutas requiere que importemos convenientemente la clase **`Request`**.

```
use Illuminate\Http\Request ;
```

Supongamos ahora que queremos añadir un parámetro adicional a la URL y, como estamos utilizando una función **callback** junto con la ruta, ésta se encargará de capturar dicho parámetro de forma automática. ¿Cómo lo hacemos?

```
Route::get("url/{parametro}", function($parametro) { ... }) ;
```

El parámetro que acabamos de definir y capturar en la función **callback** puede convivir perfectamente con el que mencionamos anteriormente:

```
Route::get("url/{parametro}", function(Request, $request, $parametro) { ... }) ;
```

También podemos definir **parámetros opcionales** utilizando el **operador Elvis** (**?**) en la definición de la ruta, para lo cual, tal y como vemos, tendremos que asignar un valor por defecto al parámetro que recibe la función **callback**.

```
Route::get("url/{parametro?}", function($parametro="valor_defecto") { ... }) ;
```

### LLAMANDO A CONTROLADORES

Aunque podemos utilizar la función **callback** en las rutas definidas en **`web.php`** o en **`api.php`**, e implementar cierta funcionalidad en el cuerpo de dicha función, lo más recomendable es seguir siempre el **patrón MVC** y llamar a un determinado método de un controlador.

```
Route::get("url", [NombreController::class, "nombre_del_metodo"]) ;
```

De esta manera, tendremos que definir en el controlador el método indicado e implementar convenientemente su funcionalidad. Imagina que queremos visualizar el perfil del usuario. Definimos la ruta:

```
Route::get("usuario/{id}", [UsuarioController::class, "perfil"]) ;
```

Y el método **`perfil`** en el controlador **`UsuarioController`**.

```
public function perfil(Request $request, $id) { ... }
```

## RESOURCES

Para poder trabajar en las respuestas de nuestras llamadas a la API con datos “preparados” o calculados un Resource de la siguiente manera:

```
php artisan make:resource NombreResource
```

*Estos apuntes fueron elaborados por Antonio Sánchez Bujaldón, modificados por Alicia Vega Moreno y se comparten aquí con su autorización y créditos correspondientes.*