IES NTRA. SRA. DE LOS REMEDIOS DE UBRIQUE

Tema 12 – Desarrollo Aplicaciones PHP - Laravel

2º DAW – Desarrollo Web Entorno Servidor

Profesor Juan Carlos Moreno

CURSO 2023/2024

Tabla de Contenido

12 Desarrollo aplicaciones PHP – Frameworks Laravel

12.1 Introducción

Laravel es un framework de código abierto para desarrollar aplicaciones y servicios con PHP 5 en adelante de manera más sencilla y ágil. Fue creado en 2011 por Taylor Otwell.

La última versión estable de Laravel es la 10.x

¿Qué es un framework?

Un Frameworks son un conjunto de utilidades o módulos que pueden usarse como base, para el desarrollo de una aplicación.

Los frameworks nos evitan escribir código repetitivo, es decir, casi todos los proyectos tienen partes comunes, como por ejemplo la conexión a la base de datos, validación de un formulario etc.. Un framework nos permite programar este código una sola vez, y extenderlo a todas nuestras aplicaciones.

Además de desarrollar más rápido nos permite utilizar buenas prácticas como por ejemplo separar la lógica de negocio de la interfaz con el usuario o hacer cosas avanzada que normalmente nos costaría mucho tiempo hacer.

Características de Laravel

Regresando a Laravel vamos a ver sus características:

- Soporte para Modelo MVC (Modelo-Vista-Controlador).
- Eloquent ORM. Maneja de una forma fácil y sencilla el manejo de bases de datos. Transforma las consultas SQL a un sistema MVC.
- Rutas. Contiene un sistema de organización y gestión de rutas que permite mantener organizado nuestro sistema.
- Middlewares. Son parecidos a controladores que se ejecutan antes y después de una petición a servidor, los cuales permiten la inserción de múltiples validaciones o procesos en las peticiones.
- Blade. Es un conjunto de platillas para crear vistas. Permite extender plantillas creadas con posibilidad de utilizar código PHP y ligarlo a bootstrap de manera que se optimice para diferentes dispositivos.
- Cache. Tiene un cache robusto el cual se puede ajustar para que la aplicación cague más rápido.

¿Cuándo utilizar Laravel?

Cuando desarrollamos a la medida podemos elegir Laravel porque necesitamos un desarrollo ágil, seguro y de fácil mantenimiento.

Si necesitamos que la web este integrada con aplicaciones como Google, servicios de email etc.

Estas pueden ser algunas razones para elegir este framework, pero cada uno puede elegir de acuerdo a las necesidades de tu proyecto.

¿Por qué Laravel?

- Reduce costos y tiempos en desarrollo.
- Fácil mantenimiento de lo creado
- La curva de aprendizaje baja comparada con otros franmeworks de PHP
- Flexible y adaptable.
- Es sencillo en el uso de datos mediante Eloquent.
- Facil uso de rutas y generaciond e URLs que ayudan a mejorar el posicionamiento web.
- Comunidad y documentación.

12.2 Patron MVC

Laravel propone en el desarrollo usar 'Routes with Closures', en lugar de un MVC tradicional con el objetivo de hacer el código más claro. Aun así permite el uso de MVC tradicional

```
<?php

      //punto de entrada de la petición HTTP
      ...
});
```

12.2.1 Modelo

Es la capa donde se trabaja con los datos, por tanto contendrá mecanismos para acceder a la información y también para actualizar su estado. Los datos los tendremos habitualmente en una base de datos, por lo que en los modelos tendremos todas las funciones que accederán a las tablas y harán los correspondientes selects, updates, inserts, etc.

No obstante, cabe mencionar que cuando se trabaja con MCV lo habitual también es utilizar otras librerías como PDO o algún ORM como Doctrine, que nos permiten trabajar con abstracción de bases de datos y persistencia en objetos. Por ello, en vez de usar directamente sentencias SQL, que suelen depender del motor de base de datos con el que se esté trabajando, se utiliza un dialecto de acceso a datos basado en clases y objetos.

Laravel incluye un sistema de mapeo de datos relacional llamado Eloquent ORM que facilita la creación de modelos. Este ORM se funda en patrón active record y su funcionamiento es muy sencillo. Es opcional el uso de Eloquent, pues también dispone de otros recursos que nos facilitan interactuar con los datos, o específicamente la creación de modelos.

La forma de crear Modelos en Laravel usando Eloquent ORM, es tan simple como:

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

class Libro extends Model {

    //defiendo el nombre de la tabla con la info de los libros
    protected $table = 'tb_libros';
}
```

12.2.2 Vista

Las vistas, como su nombre nos hace entender, contienen el código de nuestra aplicación que va a producir la visualización de las interfaces de usuario, o sea, el código que nos permitirá renderizar los estados de nuestra aplicación en HTML. En las vistas nada más tenemos los códigos HTML y PHP que nos permite mostrar la salida.

En la vista generalmente trabajamos con los datos, sin embargo, no se realiza un acceso directo a éstos. Las vistas requerirán los datos a los modelos y ellas se generará la salida, tal como nuestra aplicación requiera.

Laravel incluye de paquete un sistema de procesamiento de plantillas llamado Blade. Este sistema de plantillas favorece un código mucho más limpio en las Vistas, además de incluir un sistema de Caché que lo hace mucho más rápido. El sistema Blade de Laravel, permite una sintaxis mucho más reducida en su escritura.

Por ejemplo, en vez pintar la vista usando el código PHP:

```
<?= $mi_nombre; ?>
```

En blade se escribiría

```
{{$mi_nombre}}
```

Lo cual no es una gran ventaja sobre todo cuando siempre es posible usar una expresión resumida en PHP. No obstante, lo que si es una gran ventaja, es el modo en que Blade maneja las plantillas.

Plantillas en blade

Las plantillas en Blade son archivos de texto plano que contiene todo el HTML de la página con etiquetas que representan elementos o zonas a incluir en la plantilla, o vistas parciales como se conocen en otros frameworks PHP. Sin embargo, en Blade estos elementos incrustados se organizan en un sólo archivo. Esta es una idea muy interesante de Laravel que mejora la organización de las vistas y su rendimiento. Sobre todo cuando las vistas pueden llegar a ser muy complejas incluso con elementos anidados. En el render de una Vista completa en Laravel se usan dos archivos: la plantilla definiendo el HTML global y las zonas a incluir. Un sólo archivo, la Vista, con los elementos (partial views).

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title>@yield('titulo')</title>
</head>
<body>
        @yield('navegacion')
</body>
</html>
```

En la plantilla presentada, el código @yield() identifica al método donde como parámetro se indica el nombre de la zona desplegar. Por otro lado, el código de la vista, donde se define la plantilla a usar y la información de las distintas zonas a desplegar:

```
<!-- identificando la plantilla a utilizar -->
@extends('template')

<!-- definiendo una zona -->
@section('titulo')

@stop

<!-- definiendo otra zona -->
@section('navegacion')

@stop
```

12.2.3 Controlador

Contiene el código necesario para responder a las acciones que se solicitan en la aplicación, como visualizar un elemento, realizar una compra, una búsqueda de información, etc.

En realidad, es una capa que sirve de enlace entre las vistas y los modelos, respondiendo a los mecanismos que puedan requerirse para implementar las necesidades de nuestra aplicación. Sin embargo, su responsabilidad no es manipular directamente datos, ni mostrar ningún tipo de salida, sino servir de enlace entre los modelos y las vistas para implementar las diversas necesidades del desarrollo.

Los controladores contienen la lógica de la aplicación y permiten organizar el código en clases sin tener que escribirlo todo en las rutas. Todos los controladores deben extenderse de la clase BaseController

```
class UserController extends BaseController {
    public function mostrarPerfil($id)
    {
        $user = User::find($id);
        return View::make('user.profile', array('user' => $user));
    }
}
```

Estos pueden ser llamados en las rutas de diferentes maneras, pero la más común usándo rutas es:

```
Route::get('user/{id}', 'UserController@mostrarPerfil');
```

12.3 Primeros pasos con Laravel

12.3.1 Instalación de Laravel

Para desarrollar aplicaciones de PHP con Laravel necesitamos primero instalar y configurar un conjunto de herramientas que nos facilitan el trabajo de creación de nuevas aplicaciones. Por

un lado, requerimos tener un entorno de desarrollo en nuestro equipo que cumpla con los requerimientos del framework y por otro, es recomendable configurar y conocer las formas de acceder a una aplicación creada en dicho entorno

12.3.1.1 Preparación del entorno de desarrollo

Para desarrollar con Laravel 10.x puedes hacerlo desde Windows, Linux o MacOs siempre que tengas un servidor web con PHP 7 o superior.

A esto nosotros le llamamos entorno de desarrollo y existe una gran variedad de ellos, cada uno con un nivel de complejidad distinto al otro, desde el más básico instalando manualmente Apache o Nginx, PHP, MySQL, etc., así como instalar herramientas como XAMPP, WAMP, MAMP, etc., hasta otras más complicadas como Laravel Homestead.

Sin embargo, recomendamos las siguientes opciones para quienes estén iniciando, por su facilidad de instalación y uso:

- En Windows puedes usar: Laragon, un entorno de desarrollo para Laravel en Windows
- En Linux: Instalación de Laravel Valet en Linux
- En MacOs: Laravel Valet

12.3.1.2 Instalación composer

Por otro lado, Laravel utiliza **Composer** para manejar sus dependencias. ¿Qué significa esto? Pues el framework Laravel hace uso de una colección de paquetes o componentes propios y de terceros para agregarle funcionalidades a las aplicaciones. Por tanto, necesitamos un gestor de dependencias que se encargue automáticamente de crear proyectos, instalar, actualizar o eliminar componentes y a su vez las dependencias de éstos. Esta herramienta es Composer, el manejador de dependencias de PHP.

Para ***Windows*** puedes descargar el instalador que ofrece Composer en su sitio web que se encargará de instalarlo para que lo puedas usar en cualquier parte del sistema. Si has decidido trabajar con Laragon no necesitas realizar esta instalación, puesto que viene incluido por defecto en este entorno de desarrollo.

Puedes confirmar si tienes bien instalado Composer ejecutando en la consola desde cualquier directorio: composer y en caso de estar instalado, te mostrará un listado de todos los comandos disponibles.

12.3.1.3 Instalación de Laravel

Una vez listo el entorno de desarrollo, usaremos Composer para instalar Laravel de esta manera:

```
composer create-project --prefer-dist laravel/laravel mi-proyecto
```

Lo que significa que estamos creando un nuevo proyecto llamado mi-proyecto con el comando create-project de Composer a partir del paquete laravel/laravel, usando la opción --prefer-dist para que Composer descargue los archivos del repositorio de distribución.

Hay una alternativa para instalar Laravel y es con su instalador, que también es un paquete, por tanto, también usaremos Composer para instalarlo de forma global con el comando:

```
composer global require "laravel/installer"
```

Luego, nos tenemos que asegurar que la variable de entorno PATH del sistema operativo tenga incluido el directorio donde se alojan los paquetes instalados globalmente y así se puedan ejecutar sin ningún problema, para ello debes modificar la variable de entorno PATH para agregar la ruta:

```
C:\Users\tu-usuario\AppData\Roaming\Composer\vendor\bin
```

Bien, de esta manera ya tenemos disponible el instalador de Laravel, por tanto, podemos ejecutar desde cualquier directorio:

```
laravel new nombre-proyecto
```

y se instalará tal y como se hizo con el comando composer create-project.

12.3.2 Introducción Laravel

Empezamos creando un nuevo proyecto

Para instalar un nuevo proyecto de Laravel puedes hacerlo de 2 maneras: con el comando create-project de Composer o con el instalador laravel new como se explicó en la lección anterior. Pero hay algo más que debes saber. El instalador de Laravel solo nos permite instalar la versión actual del framework (opción por defecto) y la versión en desarrollo con el comando:

```
laravel new mi-proyecto --dev
```

Así que si quieres instalar Laravel en una versión que no sea la última versión, entonces debes usar el comando create-project que nos da la opción de poder especificar la versión que queremos usar. De esta manera:

```
composer create-project laravel/laravel nombre-proyecto "6.5”
```

El uso de un sistema de control de versiones como Git es primordial cuando quieres desarrollar de manera profesional

Por otro lado, algo clave para desarrollar eficientemente es tener un editor de texto o IDE bien configurado que nos facilite el trabajo de escribir el código de nuestra aplicación. Hay dos grandes grupos: IDE (Entorno de Desarrollo Integrado) y editores de texto. La diferencia principal es que los primeros vienen por defecto con múltiples herramientas como: autocompletado inteligente, resaltado de sintaxis, sistema de control de versiones, debugger, entre otras herramientas configuradas y listas para empezar a trabajar. En cambio, los

editores de texto son más ligeros y no vienen con todas las herramientas u opciones listas por defecto, sino que debes instalarlas y configurarlas por medio de plugins y extensiones.

Entre los IDEs para PHP tenemos a: PHPStorm, Zend Studio, Eclipse, NetBeans, Aptana Studio, etc. Entre los editores de texto están: Sublime Text, Atom, **Visual Studio Code**, NotePad++, etc. ¡Elige uno con el cual te sientas cómodo y comienza a desarrollar.

Funcionamiento Laravel

Laravel como otros frameworks usa un patron de diseño llamado *Front Controller*, lo que implica que vamos a tener un solo punto de entrada en nuestra aplicación, en nuestro caso será el archivo **index.php** que se encuentra dentro de la carpeta ***public***.

La carpeta ***public*** contendrá todos los archivos publicos de nuestra aplicación, como css, js, favicon.ico, etc... el resto de archivos y carpetas van a ser privados puesto que van a estar fuera de la carpeta ***public***, y no habrá acceso directo a estos archivos, lo cual se establece como medida de seguridad.

Dentro de la carpeta ***routes*** disponemos del archivo ***web.php***, que es donde vamos a manejar las rutas de nuestra aplicación:

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

Indica que estamos declarando una ruta de tipo *get* para la url del *home* o página principal y esta ruta a su vez retorna una vista que se llama *welcome.* Las vistas se encuentran dentro de la carpeta ***resources*** y a su vez dentro de la carpeta ***views*** donde podemos encontrar el archivo ***welcome.blade.php***, donde blade es el sistema de plantillas que usa Laravel.

Cualquier modificación que realice en dicha plantilla se reflejará en el inicio de mi aplicación Laravel.

Laravel no nos obliga a trabajar con esta vista, podemos crear una desde cero y asignarla a la ruta del home.

12.3.3 El sistema de rutas de Laravel

El sistema de Rutas de Laravel es bastante intuitivo y fácil de manejar, pero a la vez muy potente, con éste podemos crear todo tipo de rutas en la aplicación: sencillas o complejas

Las rutas son una capa muy importante en Laravel, es por ello que el Framework destina un directorio en la carpeta raíz, llamado routes, para ubicar todas las rutas de la aplicación. Por defecto, tiene 2 archivos de rutas web.php y api.php. Como sus nombres lo expresan en web.php se definen las rutas para la web y en api.php las rutas para crear APIs para la aplicación.

Podemos definir rutas de varias maneras en esta lección lo hicimos usando una función anónima, que sigue el siguiente formato:

```
Route::get('/esta-es-la-url', function () {
    return 'Hola mundo';
});
```

Se escribe la clase Route que llama al método relacionado con el verbo HTTP, en este caso, get que acepta dos parámetros: el primero es la URL que se llamará desde el navegador y el segundo es una función anónima que devuelve lo que queremos mostrar.

12.3.3.1 Rutas con parámetros

También con el sistema de rutas de Laravel puedes crear rutas más complejas que necesiten de parámetros dinámicos. Se pueden definir de la siguiente forma:

```
Route::get('/usuarios/detalles/{id}', function ($id) {
    return "Detalles del usuario: {$id}";
});
```

En este caso Laravel se encarga de capturar el segmento de la ruta que es dinámico (lo identifica porque está encerrado entre llaves). Por tanto, en la URL pasamos la identificación del parámetro encerrado entre llaves y en la función anónima lo pasamos como argumento para que pueda ser accedido y usado dentro de dicha función.

Se pueden usar tantos parámetros como sean necesarios, solo es importante que estén encerrados entre llaves {} y los nombres pueden ser alfanuméricos pero no está permitido usar el guión - pero sí el subrayado _. Además, importa el orden de los parámetros pasados a la función anónima, pero no los nombres que se les de. Por ejemplo:

```
Route::get('posts/{post_id}/comments/{comment_id}', function ($postId,
$commentId) {
    return "Este el comentario {$commentId} del post {$postId}";
});
```

12.3.3.2 Rutas con parámetros opcionales

Cuando el uso de un parámetro no es obligatorio, podemos usar el carácter ? después del nombre del parámetro para indicar que es opcional. Sin embargo, debe añadirse un valor por defecto al parámetro cuando lo colocamos en la función, por ejemplo:

```
Route::get('saludo/{name}/{nickname?}', function ($name, $nickname = null) {
    if ($nickname) {
        return "Bienvenido {$name}, tu apodo es {$nickname}";
    } else {
        return "Bienvenido {$name}, no tienes apodo";
    }
});
```

12.3.3.3 Rutas con filtros o restricciones de expresiones regulares en los parámetros

Cuando un usuario hace una petición HTTP, Laravel busca en los archivos de rutas una definición que coincida con el patrón de la URL según el método HTTP usado y en la primera coincidencia le muestra el resultado al usuario. Por tanto, el orden de precedencia de las definiciones de rutas es muy importante.

Para solucionar los posibles conflictos con el parecido en la URL de distintas rutas puedes hacerlo de 2 maneras:

- Usando el método where para agregar condiciones de expresiones regulares a la ruta. Puedes consultar nuestro tutorial Rutas con filtros en Laravel donde te explicamos detalladamente el uso del método where.
- Ordenando las rutas de tal manera que las más específicas estén al principio y las más generales al final del archivo de rutas.

Veamos el siguiente ejemplo

```
Route::get('user', function () {
    return "foo";
});

Route::post('user', function () {
    return "bar";
});
```

En este caso tenemos dos rutas hacia el mismo url /user, diferenciadas por el tipo de petición, una es de tipo POST (para enviar datos desde un formulario) y otra de tipo GET, de esta forma usando un tipo diferente indicamos cual de ellas debe encargarse de cada solicitud. Por ejemplo, una para mostrar el formulario de registro de usuarios y la otra para almacenar los datos que este formulario envié.

En las rutas también podemos recibir parámetros cómo:

```
Route::get('user/{id}', function ($id) {
    return $id;
});
```

En este caso vamos a recibir por medio de la variable $id, por ejemplo, el identificador de un usuario en la base de datos, ahora, que tal si queremos recibir como parámetro en otra ruta el nombre del usuario, pero usando el mismo slug:

```
Route::get('user/{id}', function ($id) {
    return $id;
});

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

Si tenemos dos rutas del mismo tipo, con el mismo slug (/user/) y recibiendo los parámetros bajo la misma estructura, podemos enfrentarnos a un problema, ya que probablemente todas las peticiones de tipo /user/nombre, /user/10 van a ser capturadas por la misma ruta, en este caso la primera en orden descendente.

Filtros en la ruta

Los filtros se pueden utilizar haciendo uso de expresiones regulares, por ejemplo, queremos que en el primer bloque capture solo las variables de tipo numérico ($id).

```
Route::get('user/{id}', function ($id) {
    return $id;
})->where(['id' => '[\d]+']);

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

Entonces desde ahora todas las rutas de tipo /user/10, van a ser capturadas por la primera ruta, pero ¿Qué ocurre con los datos de tipo string?, la respuesta es sencilla, podemos asegurarnos de usar la misma estructura:

```
Route::get('user/{id}', function ($id) {
    return $id;
})->where('id', '[0-9]+');

Route::get('user/{name}', function ($name) {
    return $name;
})->where(['name' => '[-\w]+']);
```

Inclusive, puedes limitar a una lista específica de posibles valores. Digamos que además del problema que ya tienes al usar user/{name} y user/{id}, tu sistema debe ser capaz de manejar posibles peticiones de tipo user/create, user/delete, /user/update. Si ingresas cualquiera de estas url, la petición va a ser capturada por la segunda ruta, pero eso no es lo que queremos, vamos a solucionarlo fácilmente.

```
Route::get('user/{id}', function ($id) {
    return $id;
})->where(['id' => '[\d]+']);

Route::get('user/{slug}', function ($slug) {
    return $slug;
})->where(['slug' => 'create|delete|update']);

Route::get('user/{name}', function ($name) {
    return $name;
})->where(['name' => '[-\w]+']);<br>
```

En este caso debemos asegurarnos de que la ruta que posee la restricción este primero para que pueda ser evaluada correctamente.

Desde ahora cuando se trate de user/delete, user/update, user/create, las peticiones van a ser capturadas por este segundo bloque ya que has definido explícitamente cuáles son los posibles valores de ese {slug} y cualquier valor de tipo string que esté fuera de esa lista va a ser capturado por la tercera ruta. Finalmente podemos probar todo esto con el siguiente código:

```
Route::get('user/{id}', function ($id) {
    return "Esto es un ID:".$id;
})->where(['id' => '[\d]+']);

Route::get('user/{slug}', function ($slug) {
    return "Esto es un Slug:".$slug;
})->where(['slug' => 'create|delete|update']);

Route::get('user/{name}', function ($name) {
    return "Esto es un Nombre:".$name;
})->where(['name' => '[-\w]+']);
```

12.3.4 Pruebas con Laravel

En el punto anterior hemos aprendido a escribir las primeras rutas de nuestra aplicación, utilizamos el navegador para probar dichas rutas y URLs. El problema de estas pruebas en el navegador es que no perduran en el tiempo ni pueden ejecutarse de forma rápida y/o automática. Hoy veremos cómo podemos probar el código que desarrollemos de forma más inteligente, utilizando el componente de pruebas automatizadas que viene incluido con Laravel.

12.3.4.1 Directorio de Pruebas

Laravel incluye en el directorio principal de tu proyecto un directorio llamado /tests. En este directorio vamos a escribir código que se va a encargar de probar el código del resto de la aplicación. Este directorio está separado en dos subdirectorios:

- El directorio Feature donde escribimos pruebas que emulan peticiones HTTP al servidor.
- El directorio Unit donde escribimos pruebas que se encargan de probar partes individuales de la aplicación (como clases y métodos).

12.3.4.2 Gestión de Pruebas

En primer lugar, tenemos que generar una prueba. para ello usamos el siguiente comando:

```
php artisan make:test NombreDeLaPruebaTest
```

Importante decir, que el nombre de todas las pruebas ha de finalizar por Test

Si queremos generar un test unitario, para un método o función concreto usamos

```
php artisan make:test NombreDeLaPruebaTest --unit
```

Para ejecutar las pruebas se usa el comando

```
Vendor/bin/phpunit
```

Pero normalmente para ejecutar las pruebas de una forma más rápida se define un alias de la siguiente forma para linux:

```
alias t=vendor/bin/phpunit
```

De estar forma si quiero ejecutar las pruebas sólo tengo que escribir el alias definido anteriormente "t".

Cuando ejecutamos las pruebas, Laravel va a recorrer todas las clases dentro del directorio *test* que tengan el sufijo *test*.

En la siguiente prueba simularemos una petición HTTP GET a la URL del módulo de usuarios. Con asserStatus comprobamos que la URL carga de forma correcta verificando que el status HTTP sea 200. Con el método assertSee comprobamos que podemos ver el texto "Usuarios":

```
/** @test */
function it_loads_the_users_list_page()
{
  $response = $this->get('/usuarios');
  $response->assertStatus(200);
  $response->assertSee('Usuarios');
}
```

Esta prueba implica que tengo creada la siguiente ruta y que todo ha funcionado correctamente.

```
Route::get('/usuarios', function () {
    return 'Usuarios';
});
```

Para que PHPUnit ejecute el método como una prueba, debes colocar la anotación /** @test */ antes de la declaración del método o colocar el prefijo test_ en el nombre del método como tal:

```
function test_it_loads_the_users_list_page
{
  //...
}
```

Veamos ahora cómo generar una nueva prueba para comprobar la siguiente ruta declarada

```
Route::get('/usuarios/{id}', function ($id) {
    return "Mostrando Detalle del usuario: {$id}";
});
```

Dentro de la nueva clase de pruebas que creamos anteriormente, añadimos el siguiente método

```
function test_it_loads_the_users_details_paga() {
      $response=$this->get('/usuarios/5');
```

```
        $response->assertStatus(200)
        $response->assertSee('Mostrando Detalle del usuario: 5');

        }
```

12.3.5 Controladores

Los controladores son un mecanismo que nos permite agrupar la lógica de peticiones HTTP relacionadas y de esta forma organizar mejor nuestro código. En esteapartado aprenderemos a hacer uso de ellos y veremos además cómo las pruebas unitarias nos permiten verificar los cambios que introducimos en nuestro código de forma fácil y rápida.

Hasta ahora la lógica de las rutas se ha incluido en el mismo archivo de ruta, donde la mayoría de las rutas están retornando sólo una cadena de texto. Si toda la lógica se incluyese en el mismo archivo podría acabar teniendo en un proyecto cientos de línea o incluso miles.

Para solucionar este problema Laravel incluye una capa llamada Controller o Controladores los cuales son básicamente un mecanismo con el cual vamos a agrupar las peticionies HTTP relacionadas dentro de una clase, y dividirla en varios métodos.

12.3.5.1 Generar un cotrolador

Generamos un nuevo controlador con el comando de Artisan make:controller pasándole el nombre que queremos darle. En el ejemplo el nombre es UserController:

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

Hecho esto, en el directorio `app/Http/Controllers` tendremos nuestro controlador `UserController`.

12.3.5.2 Métodos en un controlador

Un controlador no es más que un archivo .php con una clase que extiende de la clase `App\Http\Controllers\Controller`:

```
<?php

namespace App\Http\Controllers;

class UserController extends Controller {
     // ...
}
```

Dentro de la clase UserController agregamos nuestros métodos públicos, que después serán enlazados a una ruta:

```
public function index(){
       return 'Usuarios';
}
```

```
public function create() {
       return "Crear Nuevo Usuario";
       }

public function show($id) {
       return "Mostrando Detalle del usuario: {$id}";
       }

public function update($id) {
       return "Actualizar usuario: {$id}";
       }
```

12.3.5.3 Enlazar una ruta a un controlador

Para enlazar una ruta a un controlador pasamos como argumento el nombre del controlador y del método que queremos enlazar, separados por un `@`. En este caso queremos enlazar la ruta `/usuarios` al método `index` del controlador `UserController`:

```
// Usamos la clase userController en web.php
use App\Http\Controllers\UserController;

// Vinculamos cada ruta con un método de la clase
Route::get('/usuarios',[UserController::class, 'index']);
Route::get('/usuarios/nuevo', ,[UserController::class, 'create']);
Route::get('/usuarios/{id}, ,[UserController::class, 'show']);
Route::get('/usuarios/update/{id}',[UserController::class, 'update']);
```

12.3.5.4 Agrupación rutas de un controlador

Podemos agrupar todas las rutas pertenecientes a un mismo controlador de la siguiente forma:

```
// Usamos la clase userController en web.php
use App\Http\Controllers\UserController;

// Vinculamos cada ruta con un método de la clase
Route::controller(UserController::class)->group(function(){
       Route::get('/usuarios','index']);
       Route::get('/usuarios/nuevo', 'create']);
       Route::get('/usuarios/{id}, 'show']);
       Route::get('/usuarios/update/{id}', 'update']);
       Route::get('/usuarios/delete/{id}', 'delete']);
});
```

12.3.5.5 Crear controlador con resource

Puedo usar el parámetro *–resource* para crear un controlador con los métodos o acciones típicos de cualquier recurso, como son las acciones incluidas en un CRUD (creación, lectura, actualización y eliminación).

Veamos el siguiente ejemplo:

```
php artisan make:controller UserController –resource
```

La rutas asignadas a UserController se pueden asignar con la siguiente línea en web.php

```
use App\Http\Controllers\UserController;

Route::resource('users', UserController::class);
```

Ahora si muestro las rutas:

```
Php artisan route:list
```

Mostrará entre otras las siguientes rutas

Método	**URL**	**Action**	**Route Name**
GET	/users	index	users.index
GET	/users/create	create	users.create
POST	/users	store	users.store
GET	/users/{id}	show	users.show
GET	/users/{id}/edit	edit	users.edit
PUT/PATCH	/users/{id}	update	users.update
DELETE	/users/{id}	destroy	users.destroy

En caso de que el controlador creado con –resource no necesitase todos los métodos anteriores, podríamos establecer lar rutas de la siguiente forma:

```
use App\Http\Controllers\UserController;

Route::resource('users', PhotoController::class)->only([
    'index', 'show'
]);

Route::resource('users', PhotoController::class)->except([
    'create', 'store', 'update', 'destroy'
]);
```

12.3.5.6 Controlador de un solo método

Si quieres tener un controlador que solo tenga una acción, puedes hacerlo llamando al método `__invoke`, por ejemplo:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class WelcomUserController extends Controller
{
    public function __invoke($name, $nickname = null){

              $name =ucfirst($name);

              if ($nickname) {

                     return "Bienvenido {$name}, tu apodo es {$nickname}";

              } else {

                     return "Bienvenido {$name}";
              }

    }

}
```

En nuestra ruta ahora podemos enlazar directamente al controlador:

```
Route::get('/saludo/{name}/{nickname?}', WelcomeUserController::class);
```

Ahora si deseáramos incluir un método de prueba para el WelcomeUserController

```
public function testWelcome() {
        $this->get('/saludo')
        ->assertStatus(404);

        $this->get('/saludo/carlos')
        ->assertStatus(200)
        ->assertSee('Bienvenido Carlos);


        $this->get('/saludo/antonio/moreno)
        ->assertStatus(200)
        ->assertSee('Bienvenido Antonio, tu apodo es moreno);

}
```

12.4 Vistas en Laravel

En este apartado aprenderemos a crear plantillas completas usando Laravel y su motor de plantillas Blade. Veremos también la diferencia entre escribir plantillas con PHP plano VS Blade, así como los temas de rendimiento y de seguridad para prevenir ataques XSS.

12.4.1 Vistas

Las **vistas** son la forma de presentar el resultado (una pantalla de nuestro sitio web) de forma visual al usuario, el cual podrá interactuar con él y volver a realizar una petición. Las vistas además nos permiten separar toda la parte de presentación de resultados de la lógica (controladores) y de la base de datos (modelos). Por lo tanto no tendrán que realizar ningún tipo de consulta ni procesamiento de datos, simplemente recibirán datos y los prepararán para mostrarlos como HTML.

12.4.1.1 Definir una vista

Las vistas se almacenan en la carpeta `resources/views` como ficheros PHP, y por lo tanto tendrán la extensión `.php` o `.blade.php`. Contendrán el código HTML de nuestro sitio web, mezclado con los assets (CSS, imágenes, Javascripts, etc. que estarán almacenados en la carpeta public) y algo de código PHP (o código Blade de plantillas, como veremos más adelante) para presentar los datos de entrada como un resultado HTML.

A continuación se incluye un ejemplo de una vista simple, almacenada en el fichero `resources/views/home.php`, que simplemente mostrará por pantalla `¡Hola <nombre>!`, donde `<nombre>` es una variable de PHP que la vista tiene que recibir como entrada para poder mostrarla.

```
<!DOCTYPE html>
<html lang="es">
<head>
	<meta charset="UTF-8">
	<title>Ejemplo Vista - Laravel</title>
</head>
<body>

	<h1>¡Hola <?= $nombre; ?>!</h1>

</body>
</html>
```

12.4.1.2 Retornar una vista

Para retornar una vista retornamos el llamado a la función helper view pasando como argumento el nombre de la vista. El nombre del archivo es relativo a la carpeta `resources/views` y no es necesario indicar la extensión del archivo.

Dentro del controlador UserController el método home() estaría definido de la siguiente forma:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
```

```
class UserController extends Controller
{

    public function home() {

       return view('home');
    }
}
```

Para ello la petición url (‘/’) debe estar definida como un ruta en el archivo `web.php` de la siguiente forma:

```
Route::get('/', 'UserController@home');
```

Pero si probamos url (‘/’) nos dará un error puesto que nos informa que la variable $nombre no está definida, y es que esa variable se la tenemos que pasar como parámetro a la vista.

12.4.1.3 Pasara datos a una vista

Existen distintas formas de pasar valores a una vista:

- Arreglos asociativos
- With
- Compact

Arreglos asociativos

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

```
$users = [
    'Joel',
    'Ellie',
    'Tess',
    //...
];

return view('users', ['users' => $users]);
```

En el ejemplo anterior la vista `users` recibirá el array `$users`

Usando este método nuestro ejemplo quedaría de la siguiente forma

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{

    public function home() {

       $nombre = ‘carlos’;
```

```
        return view('home', ['nombre' => $nombre]);
    }
}
```

With

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{

    public function home() {

        $nombre = 'carlos';

        return view('home')->with('nombre', $nombre);
    }
}
```

Compact

Si los datos que queremos pasar a la vista se encuentran dentro de variables locales podemos utilizar la función `compact`, la cual acepta como argumentos los nombres de las variables y las convierte en un array asociativo:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{

    public function home() {

        $nombre = 'carlos';

        return view('home', compact('nombre'));
    }
}
```

12.4.1.4 Escapar código HTML

Laravel nos ofrece un helper llamado e que nos permite escapar HTML que podría ser insertado por los usuarios de nuestra aplicación, de manera de prevenir posibles ataques XSS:

```
<li><?php echo e($user) ?></li>
```

Ya veremos más adelante que en caso de que la vista tuviese extensión blade.php, pdríamos usar para escapar la directiva blade de la siguiente forma:

```
<li>{!!$user!!}</li>
```

12.4.2 Blade. Motor de Plantillas de Laravel

Laravel utiliza Blade para la definición de plantillas en las vistas. Esta librería permite realizar todo tipo de operaciones con los datos, además de la sustitución de secciones de las plantillas por otro contenido, herencia entre plantillas, definición de layouts o plantillas base, etc.

Los ficheros de vistas que utilizan el sistema de plantillas Blade tienen que tener la extensión `.blade.php`. Esta extensión tampoco se tendrá que incluir a la hora de referenciar una vista desde el fichero de rutas o desde un controlador. Es decir, utilizaremos `view('home')` tanto si el fichero se llama `home.php` como `home.blade.php`.

En general el código que incluye Blade en una vista empezará por los símbolos `@` o `{{`, el cual posteriormente será procesado y preparado para mostrarse por pantalla. Blade no añade sobrecarga de procesamiento, ya que todas las vistas son preprocesadas y cacheadas, por el contrario nos brinda utilidades que nos ayudarán en el diseño y modularización de las vistas.

12.4.2.1 Mostrar datos

El método más básico que tenemos en Blade es el de mostrar datos, para esto utilizaremos las llaves dobles `({{ }})` y dentro de ellas escribiremos la variable o función con el contenido a mostrar:

```
Hola {{ $name }}.
La hora actual es {{ time() }}.
```

Como hemos visto podemos mostrar el contenido de una variable o incluso llamar a una función para mostrar su resultado. Blade se encarga de escapar el resultado llamando a htmlentities para prevenir errores y ataques de tipo XSS. Si en algún caso no queremos escapar los datos tendremos que llamar a:

```
Hola {!! $name !!}
```

12.4.2.2 Mostrar un dato sólo si existe

Para comprobar que una variable existe o tiene un determinado valor podemos utilizar el operador ternario de la forma:

```
{{ isset($name) ? $name : 'Valor por defecto' }}
```

O simplemente usar la notación que incluye Blade para este fin:

```
{{ $name or 'Valor por defecto' }}
```

12.4.2.3 Comentarios

Para escribir comentarios en Blade se utilizan los símbolos {{-- y --}}, por ejemplo:

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

12.4.2.4 Bucles y estructuras condicionales

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

Estructura if:

```
@if( count($users) === 1 )
    Solo hay un usuario!
@elseif (count($users) > 1)
    Hay muchos usuarios!
@else
    No hay ningún usuario :(
@endif
```

En los siguientes ejemplos se puede ver como realizar bucles tipo **for**, **while** o **foreach**:

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

@while (true)
       <p>Soy un bucle while infinito!</p>
@endwhile

@foreach ($users as $user)
       <p>Usuario {{ $user->name }} con identificador: {{ $user->id }}</p>
@endforeach
```

En blade tambien se puede usar la directiva **@unless** que funciona como un condicional inverso

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

Con la directiva @forelse podemos asignar una opción por defecto a un ciclo foreach

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

12.4.3 Layouts con blade

A medida que nuestro proyecto crece nuestras plantillas se vuelven más complejas y es inevitable encontrarnos con que estamos repitiendo etiquetas y estructuras que podríamos compartir entre multiples vistas. Es por ello que en esta sección aprenderás a integrar cualquier diseño usando Laravel Blade; para que de esta manera puedas sacarle provecho a las diferentes directivas que ofrece este motor de plantillas y evitar así la repetición de código HTML, además de mantener tus vistas sencillas, expresivas, elegantes y bien estructuradas.

12.4.3.1 Directiva @include

Blade incluye una directiva llamada `@include`. Para usarla solamente tenemos que pasarle el nombre del archivo que queremos incluir:

```
@include('header')

      <h1>{{ $title }}</h1>
      ...
@include('footer')
```

Podemos usar múltiples directivas `@include` dentro de una misma plantilla de Blade.

12.4.3.2 Helper asset()

El helper `asset` nos dará la ruta absoluta al archivo indicado:

```
<link href="{{ asset('css/style.css') }}" rel="stylesheet">
```

Utilizando este helper podemos evitar que la ruta del archivo cambie dependiendo de la URL.

12.4.3.3 Layout principal

En lugar de separar nuestra plantilla en diferentes archivos, podemos crear una sola plantilla que tendrá toda la estructura de nuestro diseño. Podemos llamar a esta plantilla `layout.blade.php`, por ejemplo, y colocar todo el código de nuestro diseño allí. Puedes nombrar tu layout de cualquier forma, siempre y cuando coloques la extensión `.blade.php`

Utilizando la directiva `@yield` dentro de esta plantilla podemos indicar secciones (pasando como argumento el nombre de la sección) y luego en plantillas individuales podemos colocar el contenido de dichas secciones:

```
<main role="main" class="container">
    @yield('content')
</main>
```

Puedes agregar tantas directivas `@yield` como quieras a tu layout. Por ejemplo, puedes agregar una directiva yield para personalizar el título de la página:

```
<title>@yield('title') - Styde.net</title>
```

12.4.3.4 Extender una plantilla

En cada una de nuestras plantillas individuales en lugar de incluir el header o footer le indicamos a Laravel que la vista debe extender de `layout.blade.php`. No es necesario colocar la extensión del archivo. Tampoco es necesario colocar la ruta completa, ya que Laravel por defecto buscará el archivo dentro del directorio `resources/views`:

```
@extends('layout')
```

Hecho esto, debemos definir las secciones. Para ello utilizamos la directiva `@section`, pasando como argumento el nombre de la sección:

```
@section('title') Usuario {{ $id }} @endsection

@section('content')
       <!-- Contenido de la sección -->
@endsection
```

Indicamos el final o cierre de la sección con la directiva `@endsection`.

La directiva `@section` define una sección de contenido, mientras que la directiva `@yield` es usada para mostrar el contenido de una sección específica.

Dado que el titulo es una sola línea, podemos pasar el contenido como el segundo argumento de `@section`:

```
@section('title', "Usuario {$id}")
```

El código que se encuentra entre comillas es PHP y no Blade, por lo que en lugar de utilizar la sintaxis de dobles llaves `{{ $id }}` utilizaremos `{$id}` o simplemente `$id`.

12.5 Base de Datos con Laravel

En este apartado veremos una introducción al manejo de base de datos con Laravel, usaremos la consola interactiva de Laravel, llamada Tinker, para probar el constructor de consultas del framework y su ORM Eloquent, crearemos nuestras primeras tablas, insertaremos datos, realizaremos consultas, veremos una introducción al manejo de relaciones entre tablas y registros con el framework y más.

12.5.1 Configuración Inicial de la Base de Datos

Lo primero que tenemos que hacer para trabajar con bases de datos es completar la configuración. Como ejemplo vamos a configurar el acceso a una base de datos tipo MySQL. Si editamos el fichero con la configuración `config/database.php` podemos ver en primer lugar la siguiente línea:

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

En este mismo fichero de configuración, dentro de la sección connections, podemos encontrar todos los campos utilizados para configurar cada tipo de base de datos, en concreto la base de datos tipo mysql tiene los siguientes valores:

```
'mysql' => [
    'driver'    => 'mysql',
    'host'      => env('DB_HOST', 'localhost'),
    'database'  => env('DB_DATABASE', 'forge'), // Nombre de la base de datos
    'username'  => env('DB_USERNAME', 'root'), // Usuario de acceso a la bd
    'password'  => env('DB_PASSWORD', ''),      // Contraseña de acceso
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
    'strict'    => false,
],
```

Como se puede ver, básicamente los campos que tenemos que configurar para usar nuestra base de datos son: host, database, username y password. Para poner estos valores abrimos el fichero `.env` de la raíz del proyecto y los actualizamos:

```
DB_CONNECTION=mysql
DB_HOST=localhost
DB_DATABASE=maratoon
DB_USERNAME=root
DB_PASSWORD=null
```

Con este ejemplo estamos conectando a la base de datos `maratoon` en el servidor `localhost` con usuario `root` y `sin contraseña`.

12.5.2 Crer Base de Datos

Si lo que deseo es crear una bae de datos desde cero, ejecuta en comando sql correspondiente desde workbench:

```
Create database geslibros;
```

Una vez creada la base de datos para definir las tablas con sus columnas, tipos de datos restricciones y relaciones entre ellas, lo hago a través de las migraciones.

12.5.3 Migraciones

Las migraciones son un mecanismo proporcionado por Laravel con el que podemos tener una especie de control de versiones sobre los cambios en la estructura de nuestra base de datos. Con las migraciones podemos diseñar esta estructura utilizando PHP y programación orientada a objetos, sin necesidad de escribir código SQL.

La forma de funcionar de las migraciones es crear ficheros (PHP) con la descripción de la tabla a crear y posteriormente, si se quiere modificar dicha tabla se añadiría una nueva migración (un nuevo fichero PHP) con los campos a modificar. Artisan incluye comandos para crear migraciones, para ejecutar las migraciones o para hacer rollback de las mismas (volver atrás).

Por defecto las migraciones se encuentran en el directorio `database/migrations`. Cada migración es un archivo `.php` que incluye en el nombre del archivo la fecha y la hora en que fue creada la migración (en formato timestamp) y el nombre de la migración.

Por defecto en la instalación de Laravel 5 se encuentran dos migraciones ya creadas para permitir autorización y acceso al proyecto:

- create_users_table
- create_password_resets_table.

12.5.3.1 Crear Migración

Para crear una nueva migración se utiliza el comando de Artisan make:migration, al cual le pasaremos el nombre del fichero a crear y el nombre de la tabla:

```
php artisan make:migration create_articles_table --create=articles
```

Esto nos creará un fichero de migración en la carpeta `database/migrations` con el nombre `<TIMESTAMP>_create_articles_table.php`. Al añadir un timestamp a las migraciones el sistema sabe el orden en el que tiene que ejecutar (o deshacer) las mismas.

La estructura de dicho fichero será la siguiente:

```
<?php

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

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

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

Si lo que queremos es añadir una migración que modifique los campos de una tabla existente tendremos que ejecutar el siguiente comando:

```
php artisan make:migration add_descripcion_to_articles_table --table=articles
```

12.5.3.2 Métodos de una migración

Dentro de la clase de la migración encontramos dos métodos:

- `up()`
- `down()`:

En el método `up()` vamos a especificar qué queremos que haga nuestra migración. Típicamente agregaremos tablas a la base de datos, pero también podemos agregar columnas a una tabla ya existente, o incluso podemos generar una migración para eliminar una tabla o columna que ya no necesitemos.

El método `down()` tendremos que deshacer los cambios que se hagan en el up (eliminar la tabla o eliminar el campo que se haya añadido). Esto nos permitirá poder ir añadiendo y eliminando cambios sobre la base de datos y tener un control o histórico de los mismos.

12.5.3.3 Ejecutar migraciones

Después de crear una migración y de definir los campos de la tabla (en la siguiente sección veremos como especificar esto) tenemos que lanzar la migración con el siguiente comando:

```
php artisan migrate
```

Este comando aplicará la migración sobre la base de datos. Si hubiera más de una migración pendiente se ejecutarán todas. Para cada migración se llamará a su método `up` para que cree o modifique la base de datos. Posteriormente en caso de que queramos deshacer los últimos cambios podremos ejecutar:

```
php artisan migrate:rollback

# O si queremos deshacer todas las migraciones
php artisan migrate:reset
```

Un comando interesante cuando estamos desarrollando un nuevo sitio web es `migrate:refresh`, el cual deshará todos los cambios y volver a aplicar las migraciones:

```
php artisan migrate:refresh
```

Además si queremos comprobar el estado de las migraciones, para ver las que ya están instaladas y las que quedan pendientes, podemos ejecutar:

```
php artisan migrate:status
```

Acceso a la documentación oficial en Laravel.com

https://laravel.com/docs/5.7/migrations#creating-columns

12.5.3.4 Crear tablas

Normalmente para crear una tabla mediate una migración uso el siguiente comando

```
php artisan make:migration create_articles_table --create=articles
```

Es el mismo comando usado en el apartado anterior y me construye la estructura completa de la migración con el método up() y down(). Con el primer método creo la tabla y con el segundo la elimino.

Por defecto me muestra la siguiente estructura

```
Schema::create('articles', function (Blueprint $table) {
            $table->increments('id');
            $table->timestamps();
        });
```

Ahora evidentemente a esa estructura debo de añadirle el resto de columnas.

12.5.3.5 Añadir Columnas

El constructor Schema::create recibe como segundo parámetro una función que nos permite especificar las columnas que va a tener dicha tabla. En esta función podemos ir añadiendo todos los campos que queramos, indicando para cada uno de ellos su tipo y nombre, y además si queremos también podremos indicar una serie de modificadores como valor por defecto, índices, etc. Por ejemplo:

```
Schema::create('users', function($table)
{
    $table->increments('id');
    $table->string('username', 32);
    $table->string('password');
    $table->smallInteger('votos');
    $table->string('direccion');
    $table->boolean('confirmado')->default(false);
    $table->timestamps();
});
```

Tipos de campos que podemos usar:

https://laravel.com/docs/5.7/migrations#creating-columns

12.5.3.6 Añadir Índices

Comando	Descripción
$table->primary('id');	Añadir una clave primaria

Comando	Descripción
$table->primary(array('first', 'last'));	Definir una clave primaria compuesta
$table->unique('email');	Definir el campo como UNIQUE
$table->index('state');	Añadir un índice a una columna

En la tabla se especifica como añadir estos índices después de crear el campo, pero también permite indicar estos índices a la vez que se crea el campo:

```
$table->string('email')->unique();
```

12.5.3.7 Añadir Claves Ajenas

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

En este ejemplo en primer lugar añadimos la columna "user_id" de tipo UNSIGNED INTEGER (siempre tendremos que crear primero la columna sobre la que se va a aplicar la clave ajena). A continuación creamos la clave ajena entre la columna "user_id" y la columna "id" de la tabla "users".

La columna con la clave ajena tiene que ser del mismo tipo que la columna a la que apunta. Si por ejemplo creamos una columna a un índice auto-incremental tendremos que especificar que la columna sea unsigned para que no se produzcan errores.

También podemos especificar las acciones que se tienen que realizar para "on delete" y "on update":

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

12.5.3.8 Ejemplo completo base de datos GESBANK.

12.5.3.8.1 Migración para la tabla clientes

```
<?php

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

class CreateClientesTable extends Migration
{
    /**
```

```
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('clientes', function (Blueprint $table) {
            $table->increments('id');
            $table->string('apellidos', 50);
            $table->string('nombre', 25);
            $table->char('telefono',15);
            $table->string('ciudad', 25);
            $table->char('dni', 9)->unique();
            $table->string('email', 50)->unique();
            $table->timestamp('fechaalt')->useCurrent();
            $table->timestamps();
        });
    }

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

12.5.3.8.2 Migración tabla cuentas

```
<?php

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

class CreateCuentasTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('cuentas', function (Blueprint $table) {
            $table->increments('id');
            $table->char('iban', 24);
            $table->unsignedInteger('cliente_id');
            $table->timestamp('fechaAlta')->useCurrent();
            $table->decimal('saldo',10,2)->default(0);
            $table->timestamp('fechaUMov')->useCurrent();
            $table->integer('numMvtos')->default(0);
            $table->timestamps();
```

```
            $table->foreign('cliente_id')->references('id')->on('clientes')-
>onDelete('restrict')->onUpdate('cascade');
        });
    }

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

12.5.3.8.3 Migración tabla Movimientos

```
<?php

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

class CreateMovimientosTable extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::create('movimientos', function (Blueprint $table) {
            $table->increments('id');
            $table->unsignedInteger('numMovimiento');
            $table->unsignedInteger('cuenta_id');
            $table->timestamp('fechaHora')->useCurrent();
            $table->enum('tipo',['I ','R']);
            $table->text('concepto');
            $table->decimal('cantidad',10, 2);
            $table->decimal('saldo',10, 2);
            $table->timestamps();
            $table->foreign('cuenta_id')->references('id')->on('cuentas')-
>onDelete('restrict')->onUpdate('cascade');

        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
```

```
        Schema::dropIfExists('movimientos');
    }
}
```

12.5.4 Modelos y el uso de Eloquent

12.5.4.1 Modelo

El mapeado objeto-relacional (más conocido por su nombre en inglés, Object-Relational mapping, o por sus siglas ORM) es una técnica de programación para convertir datos entre un lenguaje de programación orientado a objetos y una base de datos relacional como motor de persistencia. Esto posibilita el uso de las características propias de la orientación a objetos, podremos acceder directamente a los campos de un objeto para leer los datos de una base de datos o para insertarlos o modificarlos.

Laravel incluye su propio sistema de ORM llamado Eloquent, el cual nos proporciona una manera elegante y fácil de interactuar con la base de datos. Para cada tabla de la base datos tendremos que definir su correspondiente modelo, el cual se utilizará para interactuar desde código con la tabla.ayuda a definir que tabla, atributos se pueden llenar y que otros se deben mantener ocultos.

Los modelos usan convenciones:

- El nombre de los modelos se escribe en singular, en contraste con las tablas de la BD que se escriben en plural.
- Usan notacion UpperCamelCase para sus nombres.

12.5.4.2 Definición de un Modelo

Por defecto los modelos se guardarán como clases PHP dentro de la carpeta `app`, sin embargo Laravel nos da libertad para colocarlos en otra carpeta si queremos, como por ejemplo la carpeta `app/Models`. Pero en este caso tendremos que asegurarnos de indicar correctamente el espacio de nombres.

Para definir un modelo que use Eloquent únicamente tenemos que crear una clase que herede de la clase Model:

Sin embargo es mucho más fácil y rápido crear los modelos usando el comando `make:model` de Artisan:

```
php artisan make:model User
```

12.5.4.3 Convenios en Eloquent

Nombre

En general el nombre de los modelos se pone en singular con la primera letra en mayúscula, mientras que el nombre de las tablas suele estar en plural. Gracias a esto, al definir un modelo no es necesario indicar el nombre de la tabla asociada, sino que Eloquent automáticamente buscará la tabla transformando el nombre del modelo a minúsculas y buscando su plural (en

inglés). En el ejemplo anterior que hemos creado el modelo `User` buscará la tabla de la base de datos llamada `users` y en caso de no encontrarla daría un error.

Si la tabla tuviese otro nombre lo podemos indicar usando la propiedad protegida `$table` del modelo:

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    protected $table = 'my_users';
}
```

Clave Primaria

Laravel también asume que cada tabla tiene declarada una clave primaria con el nombre `id`. En el caso de que no sea así y queramos cambiarlo tendremos que sobrescribir el valor de la propiedad protegida `$primaryKey` del modelo, por ejemplo: `protected $primaryKey = 'my_id';`.

Es importante definir correctamente este valor ya que se utiliza en determinados métodos de Eloquent, como por ejemplo para buscar registros o para crear las relaciones entre modelos.

Timestamps

Otra propiedad que en ocasiones tendremos que establecer son los timestamps automáticos. Por defecto Eloquent asume que todas las tablas contienen los campos `updated_at` y `created_at` (los cuales los podemos añadir muy fácilmente con Schema añadiendo `$table->timestamps()` en la migración). Estos campos se actualizarán automáticamente cuando se cree un nuevo registro o se modifique. En el caso de que no queramos utilizarlos (y que no estén añadidos a la tabla) tendremos que indicarlo en el modelo o de otra forma nos daría un error. Para indicar que no los actualice automáticamente tendremos que modificar el valor de la propiedad pública `$timestamps` a false, por ejemplo: `public $timestamps = false;`.

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    protected $table = 'my_users';
    protected $primaryKey = 'my_id'
    public $timestamps = false;
}
```

Fillable

Permite indicar en el modelo que campos de la tabla quiero que sean mostrados.

```
protected $fillable = [ 'name', 'email', 'password',];
```

Hiden

Permite indicar en el modelo los campos de la tabla que permanecerán ocultos

```
protected $hidden = [ 'password', 'remember_token', ];
```

Ejemplo modelo completo para la tabla **Cliente**

```
class Cliente extends Model
{
    protected $table = 'clientes';
    protected $primaryKey = 'idCliente';
    public $timestamps = false;

    protected $fillable = [
        'apellidos', 'nombre', 'telefono', 'ciudad', 'dni', 'email',
'fechaalta'
    ];
}
```

12.5.4.4 Establecer las Relaciones en el Modelo

Es muy importante indicar antes que si Laravel asume por defecto que las claves primarias van a ser `id`, las claves ajenas se van a llamar anteponiendo el nombre de la tabla seguido de un guión bajo más id, por ejemplo, podrían ser claves ajenas: `user_id`, `articulo_id`, `cliente_id`, …

Como ya sabemos podemos encontrarnos en un esquema relacional con tres tipos de relaciones:

- 1:1 - Uno a uno
- 1:N - Uno a varios
- M:N - Varios a varios

Las relaciones existentes en el esquema relacional han de definirse en el modelo perteneciente a las dos tablas que participan en la relación.

Ejemplo 1. Relación Uno a Uno.

Un usuario sólo puede tener un teléfono. Partimos para este ejemplo de que disponemos ya del modelo `User` y del modelo `Phone`.

En el modelo `User` tenemos que especificar una relación del tipo `one a one` puesto que un teléfono sólo puede pertenecer a un usuario. Para ello creamos una función pública con el nombre `phone` que se corresponde con el modelo de la tabla phones.

```
namespace App;
```

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

class User extends Model
{
    /**
     * Get the phone record associated with the user.
     */
    public function phone()
    {
        return $this->hasOne('App\Phone');
    }
}
```

Ahora hay que definir la relación `inversa` en el modelo `Phone`, usando el método `belongsTo()`

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model
{
    /**
     * Get the user that owns the phone.
     */
    public function user()
    {
        return $this->belongsTo('App\User');
    }
}
```

Ejemplo 2. Relación uno a varios

Una noticia puede tener varios comentario pero un comentario sólo puede pertenecer a una noticia. Disponemos del modelo `Post` y `Comment`.

En el modelo `Post` uso el método `hasMany()` para establecer la relación. Importante indicar que como un Post puede tener varios comentarios el nombre del la función se pone en plural `comments()`

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

Relación inversa se establece en el modelo `Post` de la siguiente forma

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get the post that owns the comment.
     */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

Ejemplo 3. Relación varios a varios

Un usuario puede tener varios roles y un rol puede estar asginado a varios usuarios.

Suponemos que tenemos creado el modelo `User` y `Role`.

En el modelo `User` usamos el método `belongsToMany()`, además el nombre de la función es en plural `roles()`, puesto que un usuario puede tener varios roles.

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The roles that belong to the user.
     */
    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }
}
```

En el modelo `Role` también usamos el método `belongsToMany()` y además el nombre de la función también la ponemos en plural, puesto que un Rol puede estar asignado a varios usuarios.

```
namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users()
```

```
    {
        return $this->belongsToMany('App\User');
    }
}
```

12.5.4.5 Modelos Base de Datos GESBANK

12.5.4.5.1 Modelo Cliente

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Cliente extends Model
{
    protected $table = "Clientes";
    protected $fillable = ['apellidos', 'nombre', 'telefono', 'ciudad', 'dni',
'email'];

    public function cuentas()
       {
              return $this->hasMany('App\Cuenta');
       }

       public function scopeSearch($query, $search) {

              return $query->where('nombre', 'LIKE', "%$search%")
                                  ->orWhere('apellidos', 'LIKE', "%$search%")
                                  ->orWhere('ciudad', 'LIKE', "%$search%")
                                  ->orWhere('email', 'LIKE', "%$search%")
                                  ->orWhere('dni', 'LIKE', "%$search%")
                                  ->orWhere('telefono', 'LIKE', "%$search%");
       }


}
```

12.5.4.5.2 Modelo Cuenta

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Cuenta extends Model
{
    protected $table = "Cuentas";
    protected $fillable = ['cliente_id','iban','saldo', 'numMvtos'];

    public function cliente()
       {
        return $this->belongsTo('App\Cliente');
```

```
       }


        public function movimientos()
        {
        return $this->hasMany('App\Movimiento');
       }

       public function scopeSearch($query, $search) {

              return $query->where('id', 'LIKE', "%$search%")
                                  ->orWhere('iban', 'LIKE', "%$search%")
                                  ->orWhere('saldo', 'LIKE', "%$search%")
                                  ->orWhere('cliente_id', 'LIKE',
"%$search%");
       }
}
```

12.5.4.5.3 Modelo Movimiento

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Movimiento extends Model
{
    protected $table = "Movimientos";
    protected $fillable =
['cuenta_id','numMovimiento','fechaHora','tipo','concepto','cantidad',
'saldo'];


     public function cuenta()
        {
        return $this->belongsTo('App\Cuenta');
        }

        public function scopeSearch($query, $search) {

              return $query->where('id', 'LIKE', "%$search%")
                                  ->orWhere('numMovimiento', 'LIKE',
"%$search%")
                                  ->orWhere('fechahora', 'LIKE', "%$search%")
                                  ->orWhere('tipo', 'IN', "('I', 'R')")
                                  ->orWhere('concepto', 'LIKE', "%$search%")
                                  ->orWhere('cantidad', 'LIKE', "%$search%")
                                  ->orWhere('saldo', 'LIKE', "%$search%")
                                  ;
       }
}
```

12.5.5 Seeders

Los Seeders por otra parte son archivos que nos van a permitir poblar nuestra base de datos para no tener que perder el tiempo escribiendo de forma manual.

Un Seeder se trata de un archivo `.php` y se ubica en la carpeta `database/seeds/` de nuestro proyecto de Laravel.

Pare crear un seeders se usa el siguiente comando:

```
php artisan make:seeder nombre_seeder
```

Creamos entonces un archivo php con el siguiente contendo

```
use Illuminate\Database\Seeder;

class GesbankSeeders extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        //
    }
}
```

Código del seeders

Ahora dentro de la función `run()` debemos añadir el código correspondiente.

Para **insertar datos**, utilizaremos el constructor de consultas SQL de Laravel. Que incluye una interfaz fluida para construir y ejecutar consultas a la base de datos. Para ello llamaremos al método `table` del Facade `DB` pasando como argumento el nombre de la tabla con la que queremos interactuar. El método insert acepta un array asociativo que reprensetará las columnas y valores que queremos guardar en la tabla.

Para utilizar el facade `DB::` debemos importar `\Illuminate\Support\Facades\DB` al principio del archivo:

```
use Illuminate\Database\Seeder;

use Illuminate\Support\Facades\DB;

class GesbankSeeders extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        DB::table('clientes')->insert([
            'apellidos' => 'García Pérez',
            'nombre' => 'José Carlos',
            'telefono' => '956789456',
```

```
            'ciudad' => 'Ubrique',
            'dni' => '12456789A',
            'email' => 'nerom@gamil.com',
            'fechaalt' => '2019-12-12'
        ]);


    }
}
```

Ahora necesito registrar `GesbankSeeders` para ello en el archivo `DatabaseSeeders` añado :

```
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        $this->call(GesbankSeeders::class);
    }
}
```

Para ejecutar un seeders

```
php artisan db:seed
```

12.5.6 Model Factories

Los model factories son una excelente forma de poblar nuestra base de datos con datos de prueba generados automáticamente.

Los model factories trabajan con el componente Faker donde se recomienda ver su documentación para su uso.

12.5.6.1 Generar Model Factory

Un model factory va siempre asociado a un modelo que previamente ya tendremos definido en nuestro proyecto Laravel.

Para poder utilizar un Model Factory necesitamos generarlo primero con el comando `make:factory`. El Model Factory será generado en el directorio `database/factories`

```
php artisan make:factory UserFactory
```

Dentro de nuestro Model Factory especificamos el atributo o los atributos que queremos generar de forma aleatoria:

```
use Faker\Generator as Faker;

$factory->define(App\User::class, function (Faker $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->unique()->safeEmail,
        'email_verified_at' => now(),
        'password' =>
'$2y$10$TKh8H1.PfQx37YgCzwiKb.KjNyWgaHb9cbcoQgdIVFlYg7B77UdFm', // secret
        'remember_token' => str_random(10),
    ];
});
```

Además es importante que indiquemos el nombre del modelo que queremos enlazar a dicho Model Factory (en este caso `\App\User::class`).

Para no tener que agregar el nombre del modelo de forma manual podemos pasar la opción `--model` al comando `make:factory`:

```
php artisan make:factory ProfessionFactory --model=User
```

12.5.6.2 Componentes Faker

El componente Faker es una librería de PHP que genera datos de prueba por nosotros

Veamos varios ejemplos

```
$faker->name;
// 'Jazmyne Romaguera'

$faker->text;
// Dolores sit sint laboriosam dolorem culpa et autem. Beatae nam sunt fugit
// et sit et mollitia sed.

$faker->cellphone;
// 9432-5656
```

Ejemplo completo faker GESBANK

Faker ***Cliente***

```
<?php

use Faker\Generator as Faker;

$factory->define(App\Cliente::class, function (Faker $faker) {
    return [
                    'apellidos' => $faker->firstName,
                    'nombre' => $faker->lastName,
                    'telefono' => $faker->e164PhoneNumber,
                    'ciudad' => $faker->city,
                    'dni' => $faker-
>regexify('\d{8}[trwagmyfpdxbnjzsqvhlcke]'),
                    'email' => $faker->unique()->safeEmail,
```

```
                    'fechaalt' => $faker->date($format = 'Y-m-d', $max =
'now'),


    ];
});
```

Faker para ***Cuentas***

```
<?php

use Faker\Generator as Faker;


$factory->define(App\Cuenta::class, function (Faker $faker) {

    $userIds = App\Cliente::all()->pluck('id')->toArray();

    return [
        'iban' => $faker->unique()->regexify('[A-Z]{2}\d{22}'),
        'cliente_id' => $faker->randomElement($userIds),
        'fechaAlta' => $faker->date($format = 'Y-m-d', $max = 'now'),
        'saldo' => $faker->randomFloat($nbMaxDecimals = 2, $min = 0, $max =
200000),
        'fechaUMov' => $faker->date($format = 'Y-m-d', $max = 'now'),
        'numMvtos' => $faker->numberBetween($min = 1, $max = 200),

    ];
});
```

Faker para ***movimientos***

```
<?php

use Faker\Generator as Faker;

$factory->define(App\Movimiento::class, function (Faker $faker) {

    $idCuentas = App\Movimiento::all()->pluck('id')->toArray();

    return [
        'cuenta_id' => $faker->randomElement($idCuentas),
        'fechaHora' => $faker->date($format = 'Y-m-d', $max = 'now'),
        'tipo' => $faker->randomElement($array = array ('I','R')),
        'cantidad' => $faker->randomFloat($nbMaxDecimals = 2, $min = 0, $max =
10000),
        'concepto' => $faker->text($maxNbChars = 50),
        'numMovimiento' =>$faker->numberBetween($min = 1, $max = 9000),
        'saldo' => $faker->randomFloat($nbMaxDecimals = 2, $min = 0, $max =
100000)

    ];
});
```

12.5.6.3 Usar Model Factory

Para utilizar un Model Factory debemos llamar al helper `factory`, especificando el nombre del modelo como argumento y finalmente encadenando el llamado al método `create`.

```
factory(User::class())->create();
```

Si lo que queremos es crear un determinado número de registros

```
factory(User::class(), 4)->create();
```

o bien de esta forma

```
factory(User::class)->times(48)->create();
```

Si queremos incluir Model Factory en un seeders de usuario sería

```
use Illuminate\Database\Seeder;
use App\User;

class ClientesSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {

        factory(User::class, 4)->create();
    }
}
```

Ahora ejecutando el seeders crearíamos 4 usuarios aleatorios en nuestra base de datos.

```
php artisan db:seed
```

12.5.6.4 Seeders Base de Datos GESBANK

12.5.6.4.1 Seeder Cliente

```
<?php

use Illuminate\Database\Seeder;
use App\Cliente;

class ClientesSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
```

```
     */
    public function run()
    {

       DB::table('clientes')->insert([
                    'apellidos' => 'García Pérez',
                    'nombre' => 'Antonio',
                    'telefono' => '956789456',
                    'ciudad' => 'Prado del Rey',
                    'dni' => '12456789B',
                    'email' => 'tamavex@gamil.com',
                    'fechaalt' => '2019-12-12',
             ]);

             DB::table('clientes')->insert([
                    'apellidos' => 'Tamayo Velázquez',
                    'nombre' => 'Pedro',
                    'telefono' => '666555989',
                    'ciudad' => 'Prado del Rey',
                    'dni' => '74932387C',
                    'email' => 'pedro@gamil.com',
                    'fechaalt' => '2018-11-12',
             ]);

         factory(Cliente::class, 20)->create();
    }
}
```

12.5.6.4.2 Seeder Cuenta

```
<?php

use Illuminate\Database\Seeder;
use App\Cuenta;

class CuentasSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {

        DB::table('cuentas')->insert([
                    'iban' => '123456',
                    'cliente_id' => '1',
                    'fechaAlta' => '1981/02/06',
                    'saldo' => '15000',
                    'fechaUMov' => '2000-02-20',
                    'numMvtos' => '12',
             ]);

             DB::table('cuentas')->insert([
                    'iban' => '323456',
                    'cliente_id' => '2',
                    'fechaAlta' => '1989-12-15',
                    'saldo' => '3000',
```

```
                    'fechaUMov' => '1999-02-10',
                    'numMvtos' => '10',
            ]);

        factory(Cuenta::class, 4)->create();
    }
}
```

Seeder Movimiento

```
<?php

use Illuminate\Database\Seeder;
use App\Movimiento;

class MovimientosSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {

        DB::table('movimientos')->insert([
                    'cuenta_id' => '1',
                    'fechaHora' => now(),
                    'tipo' => 'I',
                    'cantidad' => '2000',
                    'numMovimiento' => '1',
                    'concepto' => 'compraVehiculo',
                    'saldo' => '2000',
            ]);

            DB::table('movimientos')->insert([
                    'cuenta_id' => '1',
                    'fechaHora' => now(),
                    'tipo' => 'R',
                    'cantidad' => '100',
                    'numMovimiento' => '2',
                    'concepto' => 'ventaVehiculo',
                    'saldo' => '1900',
            ]);

            DB::table('movimientos')->insert([
                    'cuenta_id' => '2',
                    'fechaHora' => now(),
                    'tipo' => 'I',
                    'cantidad' => '50',
                    'numMovimiento' => '3',
                    'concepto' => 'cobroFactura',
                    'saldo' => '1950',
            ]);

            DB::table('movimientos')->insert([
                    'cuenta_id' => '2',
                    'fechaHora' => now(),
                    'tipo' => 'R',
                    'cantidad' => '55',
```

```
                    'numMovimiento' => '4',
                    'concepto' => 'pagoMercadona',
                    'saldo' => '1895',
            ]);


        factory(Movimiento::class, 8)->create();
    }
}
```

12.5.6.4.3 Seeder GESBANK

Este seeder será el primero que hay que llamar y permitirá vaciar todas las tablas para después regenerarlas con nuevos datos.

```
<?php

use Illuminate\Database\Seeder;

use Illuminate\Support\Facades\DB;


class GesbankSeeders extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        DB::statement('SET FOREIGN_KEY_CHECKS = 0');

        DB::table('role_user')->truncate();
        DB::table('users')->truncate();
        DB::table('roles')->truncate();

        DB::table('clientes')->truncate();
        DB::table('cuentas')->truncate();
        DB::table('movimientos')->truncate();

        DB::statement('SET FOREIGN_KEY_CHECKS = 1');



    }
}
```

12.5.6.4.4 Seeders Database

Este es el sedder principal a través del cual se llamará de forma ordenada a los anteriores.

```
<?php

use Illuminate\Database\Seeder;
```

```
class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        $this->call(GesbankSeeders::class);
        $this->call(RolesSeeders::class);
        $this->call(UsersSeeders::class);
        $this->call(ClientesSeeder::class);
        $this->call(CuentasSeeder::class);
        $this->call(MovimientosSeeder::class);
    }
}
```

12.6 Proyecto Laravel

En esta sección vamos a explicar paso a paso el proceso de elaboración de un proyecto con el frameworks de Laravel. Haciendo incapié en lo nuevo viendo de forma más rápida lo que ya está incluido en esta lección.

Partimos de que ya hemos instalado un nuevo proyecto Laravel llamado GESBANK, y que ya tenemos lista la Base de Datos GESBANK con las tablas de Clientes, Cuentas y Movimientos con sus módulos de:

- Migraciones
- Modelos
- Seeeders
- Faker

12.6.1 Controladores

Crearemos los controladores para el CRUD de la tabla de Clientes, Cuentas y Movimientos.

Para crear este tipo de controladores es más fácil usar el comando:

```
Php artisan make:controller ClienteController –resource
```

Este comando permite crear el controlador con todos métodos necesarios para realizar un CRUD de Clientes.

12.6.2 Rutas

Para crear la rutas a cada uno de los métodos de ClientesController es más fácil hacerlo de forma agrupada mediante el siguiente comando route():

```
Route::resource('clientes', 'ClienteController');
```

Este comando crea automáticamente el siguiente conjunto de rutas:

```
+--------+-----------+-------------------------+-----------------+---------------------------------------------+--------------+
| Domain | Method    | URI                     | Name            | Action                                      | Middleware   |
+--------+-----------+-------------------------+-----------------+---------------------------------------------+--------------+
|        | GET|HEAD  | /                       | index           | App\Http\Controllers\PrincipalController@index | web          |
|        | GET|HEAD  | api/user                |                 | Closure                                     | api,auth:api |
|        | GET|HEAD  | clientes                | clientes.index  | App\Http\Controllers\ClienteController@index   | web          |
|        | POST      | clientes                | clientes.store  | App\Http\Controllers\ClienteController@store   | web          |
|        | GET|HEAD  | clientes/create         | clientes.create | App\Http\Controllers\ClienteController@create  | web          |
|        | GET|HEAD  | clientes/{cliente}      | clientes.show   | App\Http\Controllers\ClienteController@show    | web          |
|        | PUT|PATCH | clientes/{cliente}      | clientes.update | App\Http\Controllers\ClienteController@update  | web          |
|        | DELETE    | clientes/{cliente}      | clientes.destroy| App\Http\Controllers\ClienteController@destroy | web          |
|        | GET|HEAD  | clientes/{cliente}/edit | clientes.edit   | App\Http\Controllers\ClienteController@edit    | web          |
+--------+-----------+-------------------------+-----------------+---------------------------------------------+--------------+
```

Fijaros en la tabla que aparece la columna name, esa columna la usaremos a partir de ahora para nombrar una ruta, por ejemplo si quiero crear un enlace para crear un cliente pondré el siguiente comando:

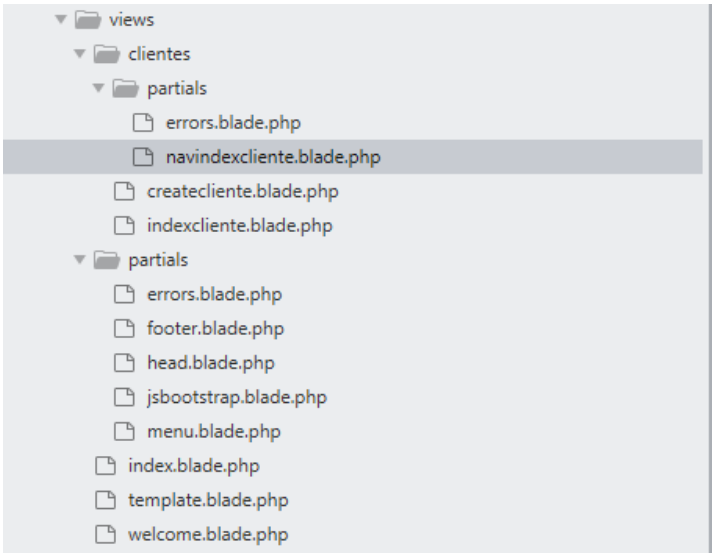
```
<a class="nav-link" href="{{ route('clientes.create') }}">Nuevo </a>
```

En el parámetro de route pongo el nombre que le he asignado a la ruta y no la URL.

12.6.3 Plantilla

Esta sección tengo que especificar la plantilla de diseño básica de mi proyecto, a partir de la cual tendré que generar las diferentes vistas, que seran extensiones de dicha plantilla.

Este proceso ya lo hemos visto en el apartado 11.4 pero hay que dejar clara la estructura de carpeta donde debo de colocar los distintos archivos de plantilla y vistas.



Si miramos la figura anterior en la carpeta `views` situaremos la plantilla de diseño base, dentro de la carpeta `partials` colocaremos las partes de dicha plantilla que serán fijas como el footer, head, …

Luego crearemos una carpeta para cada CRUD, en este ejemplo podemos ver la de `clientes` donde situaremos todas las vistas de clientes, y dentro de ella crearemos la carpeta `partials` para las secciones fijas y compartidas de todas las vistas de clientes.