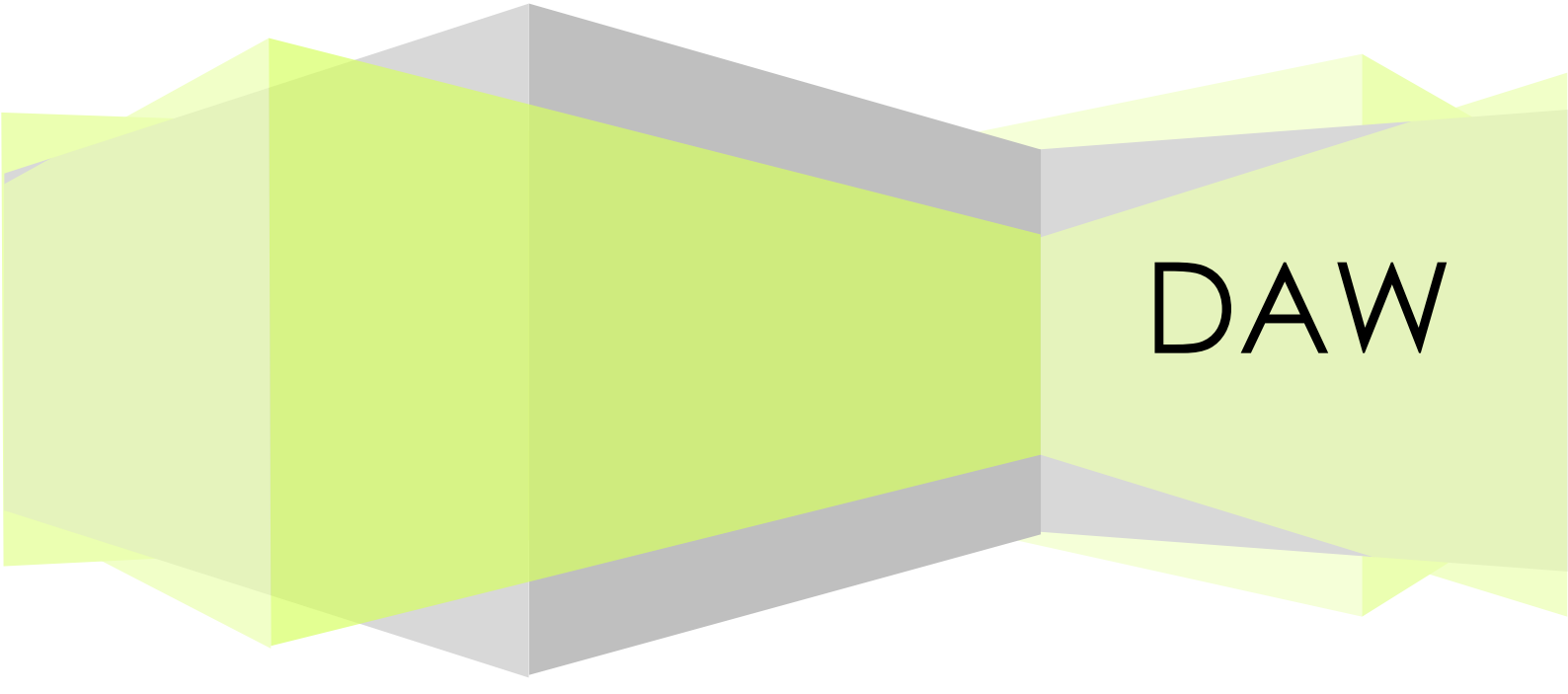
Departamento de informática y comunicaciones
Desarrollo web en entorno servidor

Programación el LARAVEL

Unidad 8

Yolanda Iglesias Suárez



Contenido

UNIDAD 8. Programación en LARAVEL 5
PARTE I. Introducción a Laravel 5
Instalación de Laravel con xampp 5
Estructura 6
FUNCIONAMIENTO BÁSICO 8
Rutas 8
Rutas básicas 8
Añadir parámetros a las rutas 9
ARTISAN 11
Vistas 12
Plantillas mediante *Blade* 14
Mostrar un dato solo si existe 14
Comentarios 14
Estructuras de control 14
Incluir una plantilla dentro de otra plantilla 15
Layouts 15
PARTE II. Controladores 16
Controlador básico 17
Crear un nuevo controlador 18
Controladores y espacios de nombres 18
Generar una URL a una acción 19
Controladores implícitos 19
Caché de rutas 20
Middleware o filtros 21
Definir un nuevo *Middleware* 21
Middleware antes o después de la petición 22
Uso de *Middleware* 23
Middleware global 23
Middleware asociado a rutas 24
Middleware dentro de controladores 25

Revisar los filtros asignados 26
Paso de parámetros 26
Rutas avanzadas 27
Middleware sobre un grupo de rutas 27
Grupos de rutas con prefijo 27
Redirecciones 28
Redirección a una acción de un controlador 29
Redirección con los valores de la petición 29
Formularios 30
Crear formularios 30
Protección contra CSRF 31
Elementos de un formulario 31
PARTE III. Bases de Datos 36
Configuración inicial 36
Configuración de la Base de Datos 36
Crear la base de datos 37
Tabla de migraciones 38
Migraciones 38
Crear una nueva migración 39
Estructura de una migración 39
Ejecutar migraciones 40
SchemaBuilder 41
Crear y borrar una tabla 41
Añadir columnas 42
Añadir índices 43
Claves ajenas 43
Inicialización de la base de datos (DatabaseSeeding) 44
Crear ficheros semilla 45
Ejecutar la inicialización de datos 46
Constructor de consultas (QueryBuilder) 46
Consultas 47
Clausula *where* 47

orderBy / groupBy / having_ 48
Offset / *Limit* 48
Transacciones 48
Más informacion 49
Modelos de datos mediante ORM 49
Definición de un modelo 49
Convenios en *Eloquent* 50
Uso de un modelo de datos 51
Consultar datos 51
Insertar datos 53
Actualizar datos 53
Borrar datos 53
Más información 53

UNIDAD 8. Programación en LARAVEL

PARTE I. Introducción a Laravel

LARAVEL es un *framework* de código abierto para desarrollar aplicaciones y servicios web con PHP 5 y PHP 7. Su filosofía es desarrollar código PHP de forma elegante y simple. Fue creado en 2011 y tiene una gran influencia de frameworks como Ruby on Rails, Sinatra y ASP.NET MVC.

Instalación de Laravel con xampp

Debemos de tener instalado:

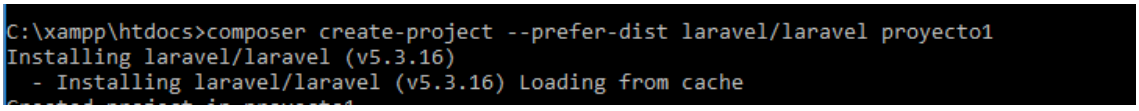
Un servidor de aplicaciones web. En este caso: **XAMPP**

Composer para descargar y gestionar las dependencias del Framework. Se trata de un administrador de dependencias para PHP que nos permite descargar paquetes desde un repositorio para agregarlo a nuestro proyecto. Por defecto, se agregan a una carpeta llamada /**vendor**. De esta manera evitamos hacer las búsquedas manualmente y el mismo Composer se puede encargar de actualizar las dependencias que hayamos descargado por una nueva versión. Para instalar Composer en Windows debemos descargarlo de su página oficial y en la sección Windows Installer, haz click en Composer-Setup.exe.

> Composer es un gestor de dependencias en proyectos, para programación en PHP. Eso quiere decir que nos permite gestionar (declarar, descargar y mantener actualizados) los paquetes de software en los que se basa nuestro proyecto PHP. Se ha convertido en una herramienta de cabecera para cualquier desarrollador en este lenguaje que aprecie su tiempo y el desarrollo ágil.

Para instalar **Laravel** los requerimientos son versión de PHP >5.4. Usando la consola/terminal vamos a la carpeta : C:/xampp/htdocs y allí escribimos:

```
composer create-project --prefer-dist laravel/laravel proyecto1
```



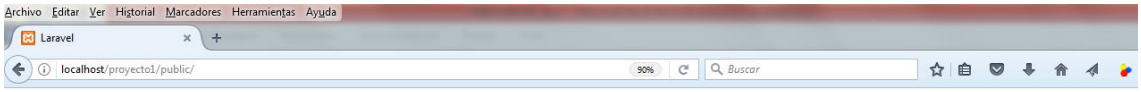
donde "proyecto 1" es el nombre de mi proyecto. Ahora *Composer* empezará a descargar e instalar el Framework y todas sus

```
C:\xampp\htdocs>composer create-project --prefer-dist laravel/laravel proyecto1
Installing laravel/laravel (v5.4.9)
  - Installing laravel/laravel (v5.4.9) Loading from cache
Created project in proyecto1
> php -r "file_exists('.env') || copy('.env.example', '.env');"
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 59 installs, 0 updates, 0 removals
  - Installing symfony/polyfill-mbstring (v1.3.0) Loading from cache
  - Installing symfony/var-dumper (v3.2.2) Loading from cache
  - Installing jakub-onderka/php-console-color (0.1) Loading from cache
  - Installing jakub-onderka/php-console-highlighter (v0.3.2) Loading from cache

  - Installing dnoegel/php-xdg-base-dir (0.1) Loading from cache
  - Installing nikic/php-parser (v3.0.3) Loading from cache
```

dependencias.

Ahora ya tendremos Laravel instalado y listo para empezar a programar con él.



Estructura

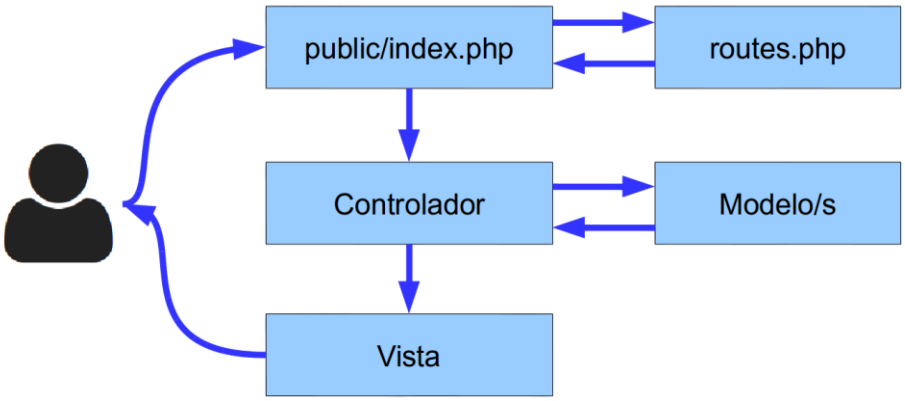
Primero que nada vamos a ver la estructura de nuestro proyecto, para así entender que hay dentro de las principales carpetas.

- **/app** - Contiene los controladores, modelos, vistas y configuraciones de la aplicación. En esta carpeta escribiremos la mayoría del código para que nuestra aplicación funcione. En la raíz tenemos un solo modelo: "user.php" que se crea por defecto
- **/app/Http.-**
  - **Controllers** son los que interaccionan con los modelos
  - **Midleware** son filtros de seguridad cuando se envía una ruta, un formulario….

- **/bootstrap**- son archivos del sistema
- **/config**.- todos los archivos de configuración del sistema
    - **App.php** tenemos los namespace para acceder a las librerías internas de Laravel,si descargamos algo nuevo para nuestra aplicación hay que instalar los namespace aquí
    - **Database.php** se configura la B.D. que ya contiene mysql, pero tb tiene sqlite y po
    - **Filesystems.php** maneja discos internos en laravel: imágenes, videos,…
- **/database**
    - Migrations: se crea la estructura para la B.D., tablas….
- **/public** - Es la única carpeta a la que los usuarios de la aplicación pueden acceder. Todo las peticiones y solicitudes a la aplicación pasan por esta carpeta, ya que en ella se encuentra el index.php, este archivo es el que inicia todo el proceso de ejecución del framework. En este directorio también se alojan los archivos CSS, Javascript, imágenes y otros archivos que se quieran hacer públicos.
- **/resources**
    - Lang : carpeta para los lenguajes
    - Views: las vistas
        - Vendor
        - Welcome.blade.php: que es la página de inicio
- **/routes**.- las rutas
    - **Web.php** es la más importante, aquí se definen las rutas para interpretar las solicitudes que el usuario hace al sistema
- **/storage**.- discos internos de laravel
- **/vendor** - En esta carpeta se alojan todas las librerías que conforman el framework y sus dependencias.
- **/lang** - En esta carpeta se guardan archivos PHP que contienen Arrays con los textos de diferentes lenguajes, en caso que se desee que la aplicación se pueda traducir.
- **/app/model**
- **/app/views** - Este directorio contiene las plantillas de HTML que usan los controladores para mostrar la información. Hay que tener en cuenta que en esta carpeta no van los Javascript, CSS o imágenes, ese tipo de archivos van alojados en la carpeta /public.

FUNCIONAMIENTO BÁSICO

El funcionamiento básico que sigue Laravel tras una petición web a una URL de nuestro sitio es el siguiente:



- Todas las peticiones entran a través del fichero public/index.php, el cual en primer lugar comprobará en el fichero de rutas (app/Http/routes.php) si la URL es válida y en caso de serlo a que controlador tiene que hacer la petición.
- A continuación se llamará al método del controlador asignado para dicha ruta. Como hemos visto, el controlador es el punto de entrada de las peticiones del usuario, el cual, dependiendo de la petición:
- Accederá a la base de datos (si fuese necesario) a través de los "modelos" para obtener datos (o para añadir, modificar o eliminar).
- Tras obtener los datos necesarios los preparará para pasárselos a la vista.
- En el tercer paso el controlador llamará a una vista con una serie de datos asociados, la cual se preparará para mostrarse correctamente a partir de los datos de entrada y por último se mostrará al usuario.

Rutas

Las rutas de nuestra aplicación aplicación se tienen que definir en el fichero *routes/web.php*. Este es el punto centralizado para la definición de rutas y cualquier ruta no definida en este fichero no será válida, generado una excepción (lo que devolverá un error 404).

Las rutas, en su forma más sencilla, pueden devolver directamente un valor desde el propio fichero de rutas, pero también podrán generar la llamada a una vista o a un controlador.

Rutas básicas

Las rutas, además de definir la URL de la petición, también indican el método con el cual se ha de hacer dicha petición. Los dos métodos más utilizados y

que empezaremos viendo son las peticiones tipo GET y tipo POST. Por ejemplo, para definir una petición tipo GET tendríamos que añadir el siguiente código a nuestro fichero `web.php`:

```
Route::get('/', function()
{
    return '¡Hola mundo!';
});
```

Este código se lanzaría cuando se realice una petición tipo GET a la ruta raíz de nuestra aplicación. Si estamos trabajando en local esta ruta sería `http://localhost` pero cuando la web esté en producción se referiría al dominio principal, por ejemplo: `http://www.dirección-de-tu-web.com`. Es importante indicar que si se realiza una petición tipo POST o de otro tipo que no sea GET a dicha dirección se devolvería un error ya que esa ruta no está definida.

Para definir una ruta tipo POST se realizaría de la misma forma pero cambiando el verbo GET por POST:

```
Route::post('foo/bar', function()
{
    return '¡Hola mundo!';
});
```

En este caso la ruta apuntaría a la dirección URL `foo/bar` (`http://localhost/foo/bar` o `http://www.dirección-de-tu-web.com/foo/bar`).

Si queremos que una ruta se defina a la vez para get y post lo podemos hacer añadiendo un array con los tipos, de la siguiente forma:

```
Route::match(array('GET', 'POST'), '/', function()
{
    return '¡Hola mundo!';
});
```

O para cualquier tipo de petición HTTP utilizando el método any:

```
Route::any('foo', function()
{
    return '¡Hola mundo!';
});
```

Añadir parámetros a las rutas

Si queremos añadir parámetros a una ruta simplemente los tenemos que indicar entre llaves {} a continuación de la ruta, de la forma:

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

```
    return 'User '.$id;
});
```

En este caso estamos definiendo la ruta `/user/{id}`, donde `id` es requerido y puede ser cualquier valor. En caso de no especificar ningún id se produciría un error. El parámetro se le pasará a la función, el cual se podrá utilizar (como veremos más adelante) para por ejemplo obtener datos de la base de datos, almacenar valores, etc.

También podemos indicar que un parámetro es opcional simplemente añadiendo el símbolo ? al final (y en este caso no daría error si no se realiza la petición con dicho parámetro):

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

// También podemos poner algún valor por defecto...

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

Laravel también permite el uso de expresiones regulares para validar los parámetros que se le pasan a una ruta. Por ejemplo, para validar que un parámetro esté formado solo por letras o solo por números:

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

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

// Si hay varios parámetros podemos validarlos usando un array:

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

A estas alturas ya tendríamos que ser capaces de crear una web con

contenido estático, simplemente modificando el fichero de rutas y devolviendo todo el contenido desde aquí. Pero esto no es lo correcto porque terminaríamos con un fichero `routes.php` inmenso con todo el código mezclado en el mismo archivo. En las siguientes secciones vamos a ver como separar el código de las vistas y más adelante añadiremos los controladores.

ARTISAN

Laravel incluye un interfaz de línea de comandos (CLI, *Command line interface*) llamado ***Artisan***. Esta utilidad nos va a permitir realizar múltiples tareas necesarias durante el proceso de desarrollo o despliegue a producción de una aplicación, por lo que nos facilitará y acelerará el trabajo.

Entre las herramientas que Laravel nos proporciona para el desarrollo de aplicaciones se encuentra Artisan, la interfaz de línea de comandos (CLI por sus siglas en inglés de *Command-line interface*), la cual es un medio para la interacción con la aplicación donde los usuarios (en este caso los desarrolladores) dan instrucciones en forma de línea de texto simple o línea de comando. Artisan está basado en el componente Console de Symfony y nos ofrece un conjunto de comandos que nos pueden ayudar a realizar diferentes tareas durante el desarrollo e incluso cuando la aplicación se encuentra en producción.

Para ver una lista de todas las opciones que incluye Artisan podemos ejecutar el siguiente comando en una consola o terminal del sistema en la carpeta raíz de nuestro proyecto:

```
php artisan list

# O simplemente:
php artisan
```

Si queremos obtener una ayuda más detallada sobre alguna de las opciones de Artisan simplemente tenemos que escribir la palabra *help* delante del comando en cuestión, por ejemplo:

```
php artisan help migrate
```

Para ver un listado con todas las rutas que hemos definido en el fichero routes.php podemos ejecutar el comando:

```
php artisan route:list
```

Esto nos mostrará una tabla con el método, la dirección, la acción y los filtros definidos para todas las rutas. De esta forma podemos comprobar todas las rutas de nuestra aplicación y asegurarnos de que esté todo correcto.

Una de las novedades de Laravel 5 es la generación de código gracias a Artisan. A través de la opción make podemos generar diferentes componentes

de Laravel (controladores, modelos, filtros, etc.) como si fueran plantillas, esto nos ahorrará mucho trabajo y podremos empezar a escribir directamente el contenido del componente. Por ejemplo, para crear un nuevo controlador tendríamos que escribir:

```
php artisan make:controller TaskController
```

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.

Las vistas se almacenan en la carpeta `resources/views` como ficheros PHP, y por lo tanto tendrán la extensión `.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.

```
<html>
    <head>
        <title>Mi Web</title>
    </head>
    <body>
        <h1>Hola <?php echo $nombre; ?>!</h1>
    </body>
</html>
```

Una vez tenemos una vista tenemos que asociarla a una ruta para poder mostrarla. Para esto tenemos que ir al fichero `routes.php` como hemos visto antes y escribir el siguiente código:

```
Route::get('/', function()
{
    return view('home', array('nombre' => 'Alberto'));
});
```

En este caso estamos definiendo que la vista se devuelva cuando se haga una petición tipo GET a la raíz de nuestro sitio. El único cambio que hemos hecho

con respecto a lo que vimos en la sección anterior de rutas ha sido en el valor devuelto por la función, el cual genera la vista usando el método `view` y la devuelve. Esta función recibe como parámetros:

- El nombre de la vista (en este caso home), el cual será un fichero almacenado en la carpeta views, acordaros que la vista anterior de ejemplo la habíamos guardado en *resources/views/home.php*. Para indicar el nombre de la vista se utiliza el mismo nombre del fichero pero sin la extensión .php.
- Como segundo parámetro recibe un *array* de datos que se le pasarán a la vista. En este caso la vista recibirá una variable llamada $nombre con valor `"Alberto"`.

Como hemos visto para referenciar una vista únicamente tenemos que escribir el nombre del fichero que la contiene pero sin la extensión `.php`. En el ejemplo, para cargar la vista almacenada en el fichero `home.php` la referenciamos mediante el nombre `home`, sin la extensión `.php` ni la ruta `resources/views`.

Las vistas se pueden organizar en sub-carpetas dentro de la carpeta `resources/views`, por ejemplo podríamos tener una carpeta `resources/views/user` y dentro de esta todas las vistas relacionadas, como por ejemplo `login.php`, `register.php` o `profile.php`. En este caso para referenciar las vistas que están dentro de sub-carpetas tenemos que utilizar la notación tipo "*dot*", en la que las barras que separan las carpetas se sustituyen por puntos. Por ejemplo, para referenciar la vista `resources/views/user/login.php` usaríamos el nombre `user.login`, o la vista `resources/views/user/register.php` la cargaríamos de la forma:

```
Route::get('register', function()
{
    return view('user.register');
});
```

Como hemos visto, para pasar datos a una vista tenemos que utilizar el segundo parámetro del método `view`, el cual acepta un array asociativo. En este array podemos añadir todas la variables que queramos utilizar dentro de la vista, ya sean de tipo variable normal (cadena, entero, etc.) u otro array o objeto con más datos. Por ejemplo, para enviar a la vista `profile` todos los datos del usuario cuyo `id` recibimos a través de la ruta tendríamos que hacer:

```
Route::get('user/profile/{id}', function($id)
{
    $user = // Cargar los datos del usuario a partir de $id
    return view('user.profile', array('user' => $user));
});
```

Plantillas mediante *Blade*

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.

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() }}.
```

Mostrar un dato solo 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' }}
```

Comentarios

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

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

Estructuras de control

Blade nos permite utilizar la estructura if de las siguientes formas:

```
@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 cómo 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
```

Esta son las estructuras de control más utilizadas. Además de estas *Blade* define algunas más que podemos ver directamente en su documentación: https://laravel.com/docs/5.3/blade

Incluir una plantilla dentro de otra plantilla

En *Blade* podemos indicar que se incluya una plantilla dentro de otra plantilla, para esto disponemos de la instrucción `@include`:

```
@include('view_name')
```

Ademas podemos pasarle un array de datos a la vista a cargar usando el segundo parámetro del método `include`:

```
@include('view_name', array('some'=>'data'))
```

Esta opción es muy útil para crear vistas que sean reutilizables o para separar el contenido de una vista en varios ficheros.

Layouts

Blade también nos permite la definición de *layouts* para crear una estructura HTML base con secciones que serán rellenadas por otras plantillas o vistas hijas. Por ejemplo, podemos crear un *layout* con el contenido principal o común de nuestra web (*head, body,* etc.) y definir una serie de secciones que serán rellenados por otras plantillas para completar el código. Este *layout* puede ser utilizado para todas las pantallas de nuestro sitio web, lo que nos permite que

en el resto de plantillas no tengamos que repetir todo este código.

A continuación se incluye un ejemplo de una plantilla tipo *layout almacenada en el fichero* `resources/views/layouts/master.blade.php`:

```
<html>
    <head>
        <title>Mi Web</title>
    </head>
    <body>
        @section('menu')
            Contenido del menu
        @show

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

Posteriormente, en otra plantilla o vista, podemos indicar que extienda el *layout* que hemos creado (con `@extends('layouts.master')`) y que complete las dos secciones de contenido que habíamos definido en el mismo:

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

@section('menu')
    @parent
    <p>Este contenido es añadido al menú principal.</p>
@endsection

@section('content')
    <p>Este apartado aparecerá en la sección "content".</p>
@endsection
```

Como se puede ver, las vistas que extienden un *layout* simplemente tienen que sobrescribir las secciones del *layout*. La directiva `@section` permite ir añadiendo contenido en las plantillas hijas, mientras que `@yield` será sustituido por el contenido que se indique. El método `@parent` carga en la posición indicada el contenido definido por el padre para dicha sección.

El método `@yield` también permite establecer un contenido por defecto mediante su segundo parámetro:

```
@yield('section', 'Contenido por defecto')
```

PARTE II. Controladores

Hasta el momento hemos visto solamente como devolver una cadena para una ruta y como asociar una vista a una ruta directamente en el fichero de rutas. Pero en general la forma recomendable de trabajar será asociar dichas

rutas a un método de un controlador. Esto nos permitirá separar mucho mejor el código y crear clases (controladores) que agrupen toda la funcionalidad de un determinado recurso. Por ejemplo, podemos crear un controlador para gestionar toda la lógica asociada al control de usuarios o cualquier otro tipo de recurso.

Como ya vimos en la sección de introducción, los controladores son el punto de entrada de las peticiones de los usuarios y son los que deben contener toda la lógica asociada al procesamiento de una petición, encargándose de realizar las consultas necesarias a la base de datos, de preparar los datos y de llamar a la vista correspondiente con dichos datos.

Controlador básico

Los controladores se almacenan en ficheros PHP en la carpeta `app/Http/Controllers` y normalmente se les añade el sufijo `Controller`, por ejemplo `UserController.php` o `MoviesController.php`. A continuación se incluye un ejemplo básico de un controlador almacenado en el fichero `app/Http/Controllers/UserController.php`:

```
<?php
namespace App\Http\Controllers;

use App\User;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
    /**
     * Mostrar información de un usuario.
     * @param  int  $id
     * @return Response
     */
    public function showProfile($id)
    {
        $user = User::findOrFail($id);
        return view('user.profile', ['user' => $user]);
    }
}
```

Todos los controladores tienen que extender la clase base `Controller`. Esta clase viene ya creada por defecto con la instalación de Laravel, la podemos encontrar en la carpeta `app/Http/Controllers`. Se utiliza para centralizar toda la lógica que se vaya a utilizar de forma compartida por los controladores de nuestra aplicación. Por defecto solo carga código para validación y autorización, pero podemos añadir en la misma todos los métodos que necesitemos.

En el código de ejemplo, el método `showProfile($id)` lo único que realiza es obtener los datos de un usuario, generar la vista `user.profile` a partir de los datos obtenidos y devolverla como valor de retorno para que se muestre por

pantalla.

Una vez definido un controlador ya podemos asociarlo a una ruta. Para esto tenemos que modificar el fichero de rutas `routes.php` de la forma:

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

En lugar de pasar una función como segundo parámetro, tenemos que escribir una cadena que contenga el nombre del controlador, seguido de una arroba @ y del nombre del método que queremos asociar. No es necesario añadir nada más, ni los parámetros que recibe el método en cuestión, todo esto se hace de forma automática.

Crear un nuevo controlador

Como hemos visto los controladores se almacenan dentro de la carpeta `app/Http/Controllers` como ficheros PHP. Para crear uno nuevo bien lo podemos hacer a mano y rellenar nosotros todo el código, o podemos utilizar el siguiente comando de Artisan que nos adelantará todo el trabajo:

```
php artisan make:controller MoviesController
```

Este comando creará el controlador `MoviesController` dentro de la carpeta `app/Http/Controllers` y lo completará con el código básico que hemos visto antes. Al añadir la opción `--plain` le indicamos que no añada ningún método al controlador, por lo que el cuerpo de la clase estará vacío. De momento vamos a utilizar esta opción para añadir nosotros mismos los métodos que necesitemos. Más adelante, cuando hablemos sobre controladores tipo RESTful, volveremos a ver esta opción.

Controladores y espacios de nombres

También podemos crear sub-carpetas dentro de la carpeta `Controllers` para organizarnos mejor. En este caso, la estructura de carpetas que creemos no tendrá nada que ver con la ruta asociada a la petición y, de hecho, a la hora de hacer referencia al controlador únicamente tendremos que hacerlo a través de su espacio de nombres.

Como hemos visto al referenciar el controlador en el fichero de rutas únicamente tenemos que indicar su nombre y no toda la ruta ni el espacio de nombres `App\Http\Controllers`. Esto es porque el servicio encargado de cargar las rutas añade automáticamente el espacio de nombres raíz para los controladores. Si metemos todos nuestros controladores dentro del mismo espacio de nombres no tendremos que añadir nada más. Pero si decidimos crear sub-carpetas y organizar nuestros controladores en sub-espacios de nombres, entonces sí que tendremos que añadir esa parte.

Por ejemplo, si creamos un controlador en:

`App\Http\Controllers\Photos\AdminController`, entonces para registrar una ruta hasta dicho controlador tendríamos que hacer:

```
Route::get('foo', 'Photos\AdminController@method');
```

Generar una URL a una acción

Para generar la URL que apunte a una acción de un controlador podemos usar el método `action` de la forma:

```
$url = action('FooController@method');
```

Por ejemplo, para crear en una plantilla con *Blade* un enlace que apunte a una acción haríamos:

```
<a href="{{ action('FooController@method') }}">¡Aprieta aquí!</a>
```

Controladores implícitos

Laravel también permite definir fácilmente la creación de controladores como recursos que capturen todas las rutas de un determinado dominio. Por ejemplo, capturar todas las consultas que se realicen a la URL "users" o "users" seguido de cualquier cosa (por ejemplo "users/profile"). Para esto en primer lugar tenemos que definir la ruta en el fichero de rutas usando `Route::controller`de la forma:

```
Route::controller('users', 'UserController');
```

Esto quiere decir que todas las peticiones realizadas a la ruta "users" o subrutas de "users" se redirigirán al controlador `UserController`. Además se capturarán las peticiones de cualquier tipo, ya sean GET o POST, a dichas rutas. Para gestionar estas rutas en el controlador tenemos que seguir un patrón a la hora de definir el nombre de los métodos: primero tendremos que poner el tipo de petición y después la sub-ruta a la que debe de responder. Por ejemplo, para gestionar las peticiones tipo GET a la URL "users/profile" tendremos que crear el método "getProfile". La única excepción a este caso es "Index" que se referirá a las peticiones a la ruta raíz, por ejemplo "getIndex" gestionará las peticiones GET a "users". A continuación se incluye un ejemplo:

```
class UserController extends BaseController
{
    public function getIndex()
    {
        //
    }
```

```
    public function postProfile()
    {
        //
    }

    public function anyLogin()
    {
        //
    }
}
```

Además, si queremos crear rutas con varias palabras lo podemos hacer usando la notación "*CamelCase*" en el nombre del método. Por ejemplo el método "getAdminProfile" será parseado a la ruta "users/admin-profile".

También podemos definir un método especial que capture las todas las peticiones "perdidas" o no capturadas por el resto de métodos. Para esto simplemente tenemos que definir un método con el nombre `missingMethod` que recibirá por parámetros la ruta y los parámetros de la petición:

```
public function missingMethod($parameters = array())
{
    //
}
```

Caché de rutas

Si definimos todas nuestras rutas para que utilicen controladores podemos aprovechar la nueva funcionalidad para crear una caché de las rutas. Es importante que estén basadas en controladores porque si definimos respuestas directas desde el fichero de rutas (como vimos en el capítulo anterior) la caché no funcionará.

Gracias a la caché Laravel indican que se puede acelerar el proceso de registro de rutas hasta 100 veces. Para generar la caché simplemente tenemos que ejecutar el comando de *Artisan*:

```
php artisan route:cache
```

Si creamos más rutas y queremos añadirlas a la caché simplemente tenemos que volver a lanzar el mismo comando. Para borrar la caché de rutas y no generar una nueva caché tenemos que ejecutar:

```
php artisan route:clear
```

Nota: La caché se recomienda crearla solo cuando ya vayamos a pasar a producción nuestra web. Cuando estamos trabajando en la web es posible que añadamos nuevas rutas y sino nos acordamos de regenerar la caché la ruta no funcionará.

Middleware o filtros

Los componentes llamados *Middleware* son un mecanismo proporcionado por Laravel para **filtrar las peticiones HTTP** que se realizan a una aplicación. Un filtro o *middleware* se define como una clase PHP almacenada en un fichero dentro de la carpeta `app/Http/Middleware`. Cada *middleware* se encargará de aplicar un tipo concreto de filtro y de decidir que realizar con la petición realizada: permitir su ejecución, dar un error o redireccionar a otra página en caso de no permitirla.

Laravel incluye varios filtros por defecto, uno de ellos es el encargado de realizar la autenticación de los usuarios. Este filtro lo podemos aplicar sobre una ruta, un conjunto de rutas o sobre un controlador en concreto. Este *middleware* se encargará de filtrar las peticiones a dichas rutas: en caso de estar logueado y tener permisos de acceso le permitirá continuar con la petición, y en caso de no estar autenticado lo redireccionará al formulario de login.

Laravel incluye *middleware* para gestionar la autenticación, el modo mantenimiento, la protección contra CSRF, y algunos mas. Todos estos filtros los podemos encontrar en la carpeta app/Http/Middleware, los cuales los podemos modificar o ampliar su funcionalidad. Pero además de estos podemos crear nuestros propios *Middleware* como veremos a continuación.

Definir un nuevo *Middleware*

Para crear un nuevo *Middleware* podemos utilizar el comando de Artisan:

```
php artisan make:middleware MyMiddleware
```

Este comando creará la clase `MyMiddleware` dentro de la carpeta `app/Http/Middleware` con el siguiente contenido por defecto:

```
<?php

namespace App\Http\Middleware;

use Closure;

class MyMiddleware
{
    /**
     * Handle an incoming request.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        return $next($request);
```

```
    }
}
```

El código generado por Artisan ya viene preparado para que podamos escribir directamente la implementación del filtro a realizar dentro de la función handle. Como podemos ver, esta función solo incluye el valor de retorno con una llamada a `return $next($request);` que lo que hace es continuar con la petición y ejecutar el método que tiene que procesarla. Como entrada la función `handle` recibe dos parámetros:

- `$request`: En la cual nos vienen todos los parámetros de entrada de la peticion.
- `$next`: El método o función que tiene que procesar la petición.

Por ejemplo podríamos crear un filtro que redirija al home si el usuario tiene menos de 18 años y en otro caso que le permita acceder a la ruta:

```
public function handle($request, Closure $next)
{
    if ($request->input('age') < 18) {
        return redirect('home');
    }

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

Como hemos dicho antes, podemos hacer tres cosas con una petición:

- Si todo es correcto permitir que la petición continúe devolviendo:

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

- Realizar una redirección a otra ruta para no permitir el acceso con:

```
return redirect('home');
```

- Lanzar una excepción o llamar al método abort para mostrar una página de error:

```
abort(403, 'Unauthorized action.');
```

Middleware antes o después de la petición

Para hacer que el código de un *Middleware* se ejecute antes o después de la petición HTTP simplemente tenemos que poner nuestro código antes o después de la llamada a `$next($request);`. Por ejemplo, el siguiente _Middleware realizaría la acción **antes** de la petición:

```
public function handle($request, Closure $next)
{
    // Código a ejecutar antes de la petición
```

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

Mientras que el siguiente *Middleware* ejecutaría el código **después** de la petición:

```
public function handle($request, Closure $next)
{
    $response = $next($request);

    // Código a ejecutar después de la petición

    return $response;
}
```

Uso de *Middleware*

De momento hemos visto para que vale y como se define un *Middleware*, en esta sección veremos cómo utilizarlos. Laravel permite la utilización de *Middleware* de tres formas distintas: global, asociado a rutas o grupos de rutas, o asociado a un controlador o a un método de un controlador. En los tres casos será necesario registrar primero el *Middleware* en la clase `app/Http/Kernel.php`.

Middleware global

Para hacer que un *Middleware* se ejecute con **todas** las peticiones HTTP realizadas a una aplicación simplemente lo tenemos que registrar en el array `$middleware` definido en la clase `app/Http/Kernel.php`. Por ejemplo:

```
protected $middleware = [

\Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,
    \App\Http\Middleware\EncryptCookies::class,
    \Illuminate\Cookie\Middleware\AddQueuedCookiesToResponse::class,
    \Illuminate\Session\Middleware\StartSession::class,
    \Illuminate\View\Middleware\ShareErrorsFromSession::class,
    \App\Http\Middleware\VerifyCsrfToken::class,
    \App\Http\Middleware\MyMiddleware::class,
];
```

En este ejemplo hemos registrado la clase *MyMiddleware* al final del array. Si queremos que nuestro *middleware* se ejecute antes que otro filtro simplemente tendremos que colocarlo antes en la posición del array.

Middleware asociado a rutas

En el caso de querer que nuestro *middleware* se ejecute solo cuando se llame a una ruta o a un grupo de rutas también tendremos que registrarlo en el fichero `app/Http/Kernel.php`, pero en el array `$routeMiddleware`. Al añadirlo a este array además tendremos que asignarle un nombre o clave, que será el que después utilizaremos asociarlo con una ruta.

En primer lugar añadimos nuestro filtro al array y le asignamos el nombre "es_mayor_de_edad":

```
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    'auth.basic' =>
\Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
    'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
    'es_mayor_de_edad' => \App\Http\Middleware\MyMiddleware::class,
];
```

Una vez registrado nuestro *middleware* ya lo podemos utilizar en el fichero de rutas `app/Http/routes.php` mediante la clave o nombre asignado, por ejemplo:

```
Route::get('dashboard', ['middleware' => 'es_mayor_de_edad', function
() {
    //...
}]);
```

En el ejemplo anterior hemos asignado el *middleware* con clave es_mayor_de_edad a la ruta dashboard. Como se puede ver se utiliza un array como segundo parámetro, en el cual indicamos el *middleware* y la acción. Si la petición supera el filtro entonces se ejecutara la función asociada.

Para asociar un filtro con una ruta que utiliza un método de un controlador se realizaría de la misma manera pero indicando la acción mediante la clave "uses":

```
Route::get('profile', [
    'middleware' => 'auth',
    'uses' => 'UserController@showProfile'
]);
```

Si queremos asociar varios *middleware* con una ruta simplemente tenemos que añadir un array con las claves. Los filtros se ejecutarán en el orden indicado en dicho array:

```
Route::get('dashboard', ['middleware' => ['auth', 'es_mayor_de_edad'],
```

```
function () {
    //...
}]);
```

Laravel también permite asociar los filtros con las rutas usando el método `middleware()` sobre la definición de la ruta de la forma:

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


// O sobre un controlador:
Route::get('profile', 'UserController@showProfile')-
>middleware('auth');
```

Middleware dentro de controladores

También es posible indicar el *middleware* a utilizar desde dentro de un controlador. En este caso los filtros también tendrán que estar registrador en el array `$routeMiddleware` del fichero `app/Http/Kernel.php`. Para utilizarlos se recomienda realizar la asignación en el constructor del controlador y asignar los filtros usando su clave mediante el método `middleware`. Podremos indicar que se filtren todos los métodos, solo algunos, o todos excepto los indicados, por ejemplo:

```
class UserController extends Controller
{
    /**
     * Instantiate a new UserController instance.
     *
     * @return void
     */
    public function __construct()
    {
        // Filtrar todos los métodos
        $this->middleware('auth');

        // Filtrar solo estos métodos...
        $this->middleware('log', ['only' => ['fooAction',
'barAction']]);

        // Filtrar todos los métodos excepto...
        $this->middleware('subscribed', ['except' => ['fooAction',
'barAction']]);
    }
}
```

Revisar los filtros asignados

Al crear una aplicación Web es importante asegurarse de que todas las rutas definidas son correctas y que las partes privadas realmente están protegidas. Para esto Laravel incluye el siguiente método de Artisan:

```
php artisan route:list
```

Este método muestra una tabla con todas las rutas, métodos y acciones. Ademas para cada ruta indica los filtros asociados, tanto si están definidos desde el fichero de rutas como **desde dentro de un controlador**. Por lo tanto es muy útil para comprobar que todas las rutas y filtros que hemos definido se hayan creado correctamente.

Paso de parámetros

Un *Middleware* también puede recibir parámetros. Por ejemplo, podemos crear un filtro para comprobar si el usuario logueado tiene un determinado rol indicado por parámetro. Para esto lo primero que tenemos que hacer es añadir un tercer parámetro a la función `handle` del *Middleware*:

```
<?php

namespace App\Http\Middleware;

use Closure;

class RoleMiddleware
{
    /**
     * Run the request filter.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \Closure  $next
     * @param  string  $role
     * @return mixed
     */
    public function handle($request, Closure $next, $role)
    {
        if (! $request->user()->hasRole($role)) {
            // No tiene el rol esperado!
        }

        return $next($request);
    }

}
```

En el código anterior de ejemplo se ha añadido el tercer parámetro `$role` a la

función. Si nuestro filtro necesita recibir más parámetros simplemente tendríamos que añadirlos de la misma forma a esta función.

Para pasar un parámetro a un *middleware* en la definición de una ruta lo tendremos que añadir a continuación del nombre del filtro separado por dos puntos, por ejemplo:

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

Si tenemos que pasar más de un parámetro al filtro los separaremos por comas, por ejemplo: `role:editor,admin`.

Rutas avanzadas

Laravel permite crear grupos de rutas para especificar opciones comunes a todas ellas, como por ejemplo un *middleware*, un prefijo, un subdominio o un espacio de nombres que se tiene que aplicar sobre todas ellas.

A continuación vamos a ver algunas de estas opciones, en todos los casos usaremos el método `Route::group`, el cual recibirá como primer parámetro las opciones a aplicar sobre todo el grupo y como segundo parámetro una clausula con la definición de las rutas.

Middleware sobre un grupo de rutas

Esta opción es muy útil para aplicar un filtro sobre todo un conjunto de rutas, de esta forma solo tendremos que especificar el filtro una vez y además nos permitirá dividir las rutas en secciones (distinguiendo mejor a que secciones se les está aplicando un filtro):

```
Route::group(['middleware' => 'auth'], function () {
    Route::get('/', function ()    {
        // Ruta filtrada por el middleware
    });

    Route::get('user/profile', function () {
        // Ruta filtrada por el middleware
    });
});
```

Grupos de rutas con prefijo

También podemos utilizar la opción de agrupar rutas para indicar un prefijo que se añadirá a todas las URL del grupo. Por ejemplo, si queremos definir una

sección de rutas que empiecen por el prefijo `dashboard` tendríamos que hacer lo siguiente:

```
Route::group(['prefix' => 'dashboard'], function () {
    Route::get('catalog', function () { /* ... */ });
    Route::get('users', function () { /* ... */ });
});
```

También podemos crear grupos de rutas dentro de otros grupos. Por ejemplo para definir un grupo de rutas a utilizar en una API y crear diferentes rutas según la versión de la API podríamos hacer:

```
Route::group(['prefix' => 'api'], function()
{
    Route::group(['prefix' => 'v1'], function()
    {
        // Rutas con el prefijo api/v1
        Route::get('recurso',      'ControllerAPIv1@getRecurso');
        Route::post('recurso',     'ControllerAPIv1@postRecurso');
        Route::get('recurso/{id}', 'ControllerAPIv1@putRecurso');
    });

    Route::group(['prefix' => 'v2'], function()
    {
        // Rutas con el prefijo api/v2
        Route::get('recurso',      'ControllerAPIv2@getRecurso');
        Route::post('recurso',     'ControllerAPIv2@postRecurso');
        Route::get('recurso/{id}', 'ControllerAPIv2@putRecurso');
    });
});
```

De esta forma podemos crear secciones dentro de nuestro fichero de rutas para agrupar, por ejemplo, todas las rutas públicas, todas las de la sección privada de administración, sección privada de usuario, las rutas de las diferentes versiones de la API de nuestro sitio, etc.

Esta opción también la podemos aprovechar para especificar parámetros comunes que se recogerán para todas las rutas y se pasarán a todos los controladores o funciones asociadas, por ejemplo:

```
Route::group(['prefix' => 'accounts/{account_id}'], function () {
    Route::get('detail', function ($account_id)  { /* ... */ });
    Route::get('settings', function ($account_id)  { /* ... */ });
});
```

Redirecciones

Como respuesta a una petición también podemos devolver una redirección. Esta opción será interesante cuando, por ejemplo, el usuario no esté logueado y lo queramos redirigir al formulario de login, o cuando se produzca un error en

la validación de una petición y queramos redirigir a otra ruta.

Para esto simplemente tenemos que utilizar el método redirect indicando como parámetro la ruta a redireccionar, por ejemplo:

```
return redirect('user/login');
```

O si queremos volver a la ruta anterior simplemente podemos usar el método `back`:

```
return back();
```

Redirección a una acción de un controlador

También podemos redirigir a un método de un controlador mediante el método `action` de la forma:

```
return redirect()->action('HomeController@index');
```

Si queremos añadir parámetros para la llamada al método del controlador tenemos que añadirlos pasando un array como segundo parámetro:

```
return redirect()->action('UserController@profile', [1]);
```

Redirección con los valores de la petición

Las redirecciones se suelen utilizar tras obtener algún error en la validación de un formulario o tras procesar algunos parámetros de entrada. En este caso, para que al mostrar el formulario con los errores producidos podamos añadir los datos que había escrito el usuario tendremos que volver a enviar los valores enviados con la petición usando el método `withInput()`

```
return redirect('form')->withInput();

// O para reenviar los datos de entrada excepto algunos:
return redirect('form')->withInput($request->except('password'));
```

Este método también lo podemos usar con la función `back` o con la función `action`:

```
return back()->withInput();

return redirect()->action('HomeController@index')->withInput();
```

Formularios

La última versión de Laravel no incluye ninguna utilidad para la generación de formularios. En esta sección vamos a repasar brevemente como crear un formulario usando etiquetas de HTML, los distintos elementos o *inputs* que podemos utilizar, además también veremos cómo conectar el envío de un formulario con un controlador, como protegernos de ataques CSRF y algunas cuestiones más.

Crear formularios

Para abrir y cerrar un formulario que apunte a la URL actual y utilice el método POST tenemos que usar las siguientes etiquetas HTML:

```
<form method="POST">
    ...
</form>
```

Si queremos cambiar la URL de envío de datos podemos utilizar el atributo `action` de la forma:

```
<form action="{{ url('foo/bar') }}" method="POST">
    ...
</form>
```

La función `url` generará la dirección a la ruta indicada. Ademas también podemos usar la función `action` para indicar directamente el método de un controlador a utilizar, por ejemplo: `action('HomeController@getIndex')`

Como hemos visto anteriormente, en Laravel podemos definir distintas acciones para procesar peticiones realizadas a una misma ruta pero usando un método distinto (GET, POST, PUT, DELETE). Por ejemplo, podemos definir la ruta "user" de tipo GET para que nos devuelva la página con el formulario para crear un usuario, y por otro lado definir la ruta "user" de tipo POST para procesar el envío del formulario. De esta forma cada ruta apuntará a un método distinto de un controlador y nos facilitará la separación del código.

HTML solo permite el uso de formularios de tipo GET o POST. Si queremos enviar un formulario usando otros de los métodos (o verbos) definidos en el protocolo REST, como son PUT, PATCH o DELETE, tendremos que añadir un campo oculto para indicarlo. Laravel establece el uso del nombre "_method" para indicar el método a usar, por ejemplo:

```
<form action="/foo/bar" method="POST">
```

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

Laravel se encargará de recoger el valor de dicho campo y de procesarlo como una petición tipo PUT (o la que indiquemos). Además, para facilitar más la definición de este tipo de formularios ha añadido la función `method_field` que directamente creará este campo oculto:

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

Protección contra CSRF

El CSRF (del inglés *Cross-site request forgery* o falsificación de petición en sitios cruzados) es un tipo de exploit malicioso de un sitio web en el que comandos no autorizados son transmitidos por un usuario en el cual el sitio web confía.

Laravel proporciona una forma fácil de protegernos de este tipo de ataques. Simplemente tendremos que llamar al método `csrf_field` después de abrir el formulario, igual que vimos en la sección anterior, este método añadirá un campo oculto ya configurado con los valores necesarios. A continuación se incluye un ejemplo de uso:

```
<form action="/foo/bar" method="POST">
    {{ csrf_field() }}
    ...
</form>
```

Elementos de un formulario

A continuación vamos a ver los diferentes elementos que podemos añadir a un formulario. En todos los tipos de campos en los que tengamos que recoger datos es importante añadir sus atributos `name` e `id`, ya que nos servirán después para recoger los valores rellenados por el usuario.

Campos de texto

Para crear un campo de texto usamos la etiqueta de HTML `input`, para la cual tenemos que indicar el tipo `text` y su nombre e identificador de la forma:

```
<input type="text" name="nombre" id="nombre">
```

En este ejemplo hemos creado un campo de texto vacío cuyo nombre e identificador es "nombre". El atributo `name` indica el nombre de variable donde

se guardará el texto introducido por el usuario y que después utilizaremos desde el controlador para acceder al valor.

Si queremos podemos especificar un valor por defecto usando el atributo `value`:

```
<input type="text" name="nombre" id="nombre" value="Texto inicial">
```

Desde una vista con *Blade* podemos asignar el contenido de una variable (en el ejemplo `$nombre`) para que aparezca el campo de texto con dicho valor. Esta opción es muy útil para crear formularios en los que tenemos que editar un contenido ya existente, como por ejemplo editar los datos de usuario. A continuación se muestra un ejemplo:

```
<input type="text" name="nombre" id="nombre" value="{{ $nombre }}">
```

Para mostrar los valores introducidos en una petición anterior podemos usar el método `old`, el cual recuperará las variables almacenadas en la petición anterior. Por ejemplo, imaginad que creáis un formulario para el registro de usuarios y al enviar el formulario comprobáis que el usuario introducido está repetido. En ese caso se tendría que volver a mostrar el formulario con los datos introducidos y marcar dicho campo como erróneo. Para esto, después de comprobar que hay un error en el controlador, habría que realizar una redirección a la página anterior añadiendo la entrada como ya vimos con `withInput()`, por ejemplo: `return back()->withInput();`. El método `withInput()` añade todas las variables de entrada a la sesión, y esto nos permite recuperarlas después de la forma:

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

Más adelante, cuando veamos como recoger los datos de entrada revisaremos el proceso completo para procesar un formulario.

Más campos tipo input

Utilizando la etiqueta `input` podemos crear más tipos de campos como contraseñas o campos ocultos:

```
<input type="password" name="password" id="password">

<input type="hidden" name="oculto" value="valor">
```

Los campos para contraseñas lo único que hacen es ocultar las letras escritas.

Los campos ocultos se suelen utilizar para almacenar opciones o valores que se desean enviar junto con los datos del formulario pero que no se tienen que mostrar al usuario. En las secciones anteriores ya hemos visto que Laravel lo utiliza internamente para almacenar un *hash* o código para la protección contra ataques tipo CSRF y que también lo utiliza para indicar si el tipo de envío del formulario es distinto de POST o GET. Además nosotros lo podemos utilizar para almacenar cualquier valor que después queramos recoger justo con los datos del formulario.

También podemos crear otro tipo de *inputs* como *email*, *number*, *tel*, etc. (podéis consultar la lista de tipos permitidos aquí: http://www.w3schools.com/html/html_form_input_types.asp).

Para definir estos campos se hace exactamente igual que para un campo de texto pero cambiando el tipo por el deseado, por ejemplo:

```
<input type="email" name="correo" id="correo">

<input type="number" name="numero" id="numero">

<input type="tel" name="telefono" id="telefono">
```

Textarea

Para crear un área de texto simplemente tenemos que usar la etiqueta HTML `textarea` de la forma:

```
<textarea name="texto" id="texto"></textarea>
```

Esta etiqueta además permite indicar el número de filas (rows) y columnas (cols) del área de texto. Para insertar un texto o valor inicial lo tenemos que poner entre la etiqueta de apertura y la de cierre. A continuación se puede ver un ejemplo completo:

```
<textarea name="texto" id="texto" rows="4" cols="50">Texto por
defecto</textarea>
```

Etiquetas

Las etiquetas nos permiten poner un texto asociado a un campo de un formulario para indicar el tipo de contenido que se espera en dicho campo. Por ejemplo añadir el texto "Nombre" antes de un input tipo texto donde el usuario tendrá que escribir su nombre.

Para crear una etiqueta tenemos que usar el tag "`label`" de HTML:

```
<label for="nombre">Nombre</label>
```

Donde el atributo `for` se utiliza para especificar el identificador del campo relacionado con la etiqueta. De esta forma, al pulsar sobre la etiqueta se marcará automáticamente el campo relacionado. A continuación se muestra un ejemplo completo:

```
<label for="correo">Correo electrónico:</label>
<input type="email" name="correo" id="correo">
```

Checkbox y Radio buttons

Para crear campos tipo *checkbox* o tipo *radio button* tenemos que utilizar también la etiqueta `input`, pero indicando el tipo `chekbox` o `radio` respectivamente. Por ejemplo, para crear un *checkbox* para aceptar los términos escribiríamos:

```
<label for="terms">Aceptar términos</label>
<input type="checkbox" name="terms" id="terms" value="1">
```

En este caso, al enviar el formulario, si el usuario marca la casilla nos llegaría la variable con nombre `terms` con valor 1. En caso de que no marque la casilla no llegaría nada, ni siquiera la variable vacía.

Para crear una lista de *checkbox* o de *radio button* es importante que todos tengan el **mismo nombre** (para la propiedad name). De esta forma los valores devueltos estarán agrupados en esa variable, y además, el *radio button* funcionará correctamente: al apretar sobre una opción se desmarcará la que esté seleccionada en dicho grupo (entre todos los que tengan el mismo nombre). Por ejemplo:

```
<label for="color">Elige tu color favorito:</label>
<br>
<input type="radio" name="color" id="color" value="rojo">Rojo<br>
<input type="radio" name="color" id="color" value="azul">Azul<br>
<input type="radio" name="color" id="color"
value="amarillo">Amarillo<br>
<input type="radio" name="color" id="color" value="verde">Verde<br>
```

Además podemos añadir el atributo `checked` para marcar una opción por defecto:

```
<label for="clase">Clase:</label>
<input type="radio" name="clase" id="clase" value="turista"
checked>Turista<br>
<input type="radio" name="clase" id="clase"
value="preferente">Preferente<br>
```

Ficheros

Para generar un campo para subir ficheros utilizamos también la etiqueta *input* indicando en su tipo el valor `file`, por ejemplo:

```
<label for="imagen">Sube la imagen:</label>
<input type="file" name="imagen" id="imagen">
```

Para enviar ficheros la etiqueta de apertura del formulario tiene que cumplir dos requisitos importantes:

- El método de envío tiene que ser POST o PUT.
- Tenemos que añadir el atributo *enctype="multipart/form-data"* para indicar la codificación.

A continuación se incluye un ejemplo completo:

```
<form enctype="multipart/form-data" method="post">
    <label for="imagen">Sube la imagen:</label>
    <input type="file" name="imagen" id="imagen">
</form>
```

Listas desplegables

Para crear una lista desplegable utilizamos la etiqueta HTML `select`. Las opciones la indicaremos entre la etiqueta de apertura y cierre usando elementos `option`, de la forma:

```
<select name="marca">
  <option value="volvo">Volvo</option>
  <option value="saab">Saab</option>
  <option value="mercedes">Mercedes</option>
  <option value="audi">Audi</option>
</select>
```

En el ejemplo anterior se creará una lista desplegable con cuatro opciones. Al enviar el formulario el valor seleccionado nos llegará en la variable `marca`. Además, para elegir una opción por defecto podemos utilizar el atributo `selected`, por ejemplo:

```
<label for="talla">Elige la talla:</label>
<select name="talla" id="talla">
  <option value="XS">XS</option>
  <option value="S">S</option>
  <option value="M" selected>M</option>
  <option value="L">L</option>
  <option value="XL">XL</option>
```

```
</select>
```

Botones

Por último vamos a ver como añadir botones a un formulario. En un formulario podremos añadir tres tipos distintos de botones:

- `submit` para enviar el formulario,
- `reset` para restablecer o borrar los valores introducidos y
- `button` para crear botones normales para realizar otro tipo de acciones (como volver a la página anterior).

A continuación se incluyen ejemplo de cada uno de ellos:

```
<button type="submit">Enviar</button>
<button type="reset">Borrar</button>
<button type="button">Volver</button>
```

PARTE III. Bases de Datos

Laravel facilita la configuración y el uso de diferentes tipos de base de datos: MySQL, Postgres, SQLite y SQL Server. En el fichero de configuración (`config/database.php`) tenemos que indicar todos los parámetros de acceso a nuestras bases de datos y además especificar cuál es la conexión que se utilizará por defecto. En Laravel podemos hacer uso de varias bases de datos a la vez, aunque sean de distinto tipo. Por defecto se accederá a la que especifiquemos en la configuración y si queremos acceder a otra conexión lo tendremos que indicar expresamente al realizar la consulta.

En este capítulo veremos cómo configurar una base de datos, como crear tablas y especificar sus campos desde código, como inicializar la base de datos y como construir consultas tanto de forma directa como a través del ORM llamado*Eloquent*.

Configuración inicial

En este primer apartado vamos a ver los primeros pasos que tenemos que dar con Laravel para empezar a trabajar con bases de datos. Para esto vamos a ver a continuación como definir la configuración de acceso, como crear una base de datos y como crear la tabla de migraciones, necesaria para crear el resto de tablas.

Configuración 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'),
```

Este valor indica el tipo de base de datos a utilizar por defecto. Como vimos en el primer capítulo Laravel utiliza el sistema de variables de entorno para separar las distintas configuraciones de usuario o de máquina. El método `env('DB_CONNECTION', 'mysql')` lo que hace es obtener el valor de la variable `DB_CONNECTION` del fichero `.env`. En caso de que dicha variable no esté definida devolverá el valor por defecto `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', 'forge'), // 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*. El *host* lo podemos dejar como está si vamos a usar una base de datos local, mientras que los otros tres campos sí que tenemos que actualizarlos con el nombres de la base de datos a utilizar y el usuario y la contraseña de acceso. Para poner estos valores abrimos el fichero `.env` de la raíz del proyecto y los actualizamos:

```
DB_CONNECTION=mysql
DB_HOST=localhost
DB_DATABASE=nombre-base-de-datos
DB_USERNAME=nombre-de-usuario
DB_PASSWORD=contraseña-de-acceso
```

Crear la base de datos

Para crear la base de datos que vamos a utilizar en MySQL podemos utilizar la herramienta *PHPMyAdmin* que se ha instalado con el paquete XAMPP. Para

esto accedemos a la ruta:

```
http://localhost/phpmyadmin
```

La cual nos mostrará un panel para la gestión de las bases de datos de MySQL, que nos permite, además de realizar cualquier tipo de consulta SQL, crear nuevas bases de datos o tablas, e insertar, modificar o eliminar los datos directamente. En nuestro caso apretamos en la pestaña "Bases de datos" y creamos una nueva base de datos. El nombre que le pongamos tiene que ser el mismo que el que hayamos indicado en el fichero de configuración de Laravel.

Tabla de migraciones

A continuación vamos a crear la tabla de migraciones. Laravel utiliza las migraciones para poder definir y crear las tablas de la base de datos desde código, y de esta manera tener un control de las versiones de las mismas.

Para poder empezar a trabajar con las migraciones es necesario en primer lugar crear la tabla de migraciones. Para esto tenemos que ejecutar el siguiente comando de Artisan:

```
phpartisanmigrate:install
```

Nota: Si nos diese algún error tendremos que revisar la configuración que hemos puesto de la base de datos y si hemos creado la base de datos con el nombre, usuario y contraseña indicado.

Si todo funciona correctamente ahora podemos ir al navegador y acceder de nuevo a nuestra base de datos con PHPMyAdmin, podremos ver que se nos habrá creado la tabla `migrations`. Con esto ya tenemos configurada la base de datos y el acceso a la misma. En las siguientes secciones veremos cómo añadir tablas y posteriormente como realizar consultas.

Migraciones

Las migraciones son un sistema de control de versiones para bases de datos. Permiten que un equipo trabaje sobre una base de datos añadiendo y modificando campos, manteniendo un histórico de los cambios realizados y del estado actual de la base de datos. Las migraciones se utilizan de forma conjunta con la herramienta *Schemabuilder* (que veremos en la siguiente sección) para gestionar el esquema de base de datos de la aplicación.

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).

Crear una nueva 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:migrationcreate_users_table --create=users
```

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

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:migrationadd_votes_to_user_table --table=users
```

En este caso se creará también un fichero en la misma carpeta, con el nombre

```
<TIMESTAMP>_add_votes_to_user_table.php
```

pero preparado para modificar los campos de dicha tabla.

Por defecto, al indicar el nombre del fichero de migraciones se suele seguir siempre el mismo patrón (aunque el realidad el nombre es libre). Si es una migración que crea una tabla el nombre tendrá que ser

```
create_<table-name>_table
```

y si es una migración que modifica una tabla será

```
<action>_to_<table-name>_table
```

Estructura de una migración

El fichero o clase PHP generada para una migración siempre tiene una estructura similar a la siguiente:

```
<?php

useIlluminate\Database\Schema\Blueprint;
```

```
useIlluminate\Database\Migrations\Migration;

classCreateUsersTableextendsMigration
{
/**
     * Run the migrations.
     * @return void
     */
publicfunctionup()
{
//
    }

/**
     * Reverse the migrations.
     * @return void
     */
publicfunctiondown()
{
//
    }
}
```

En el método `up` es donde tendremos crear o modificar la tabla, y en 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.

Ejecutar migraciones

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

```
phpartisanmigrate
```

Nota: Si nos aparece el error "classnotfound" lo podremos solucionar llamando a `composerdump-autoload` y volviendo a lanzar las migraciones.

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:

```
phpartisanmigrate:rollback

# O si queremos deshacer todas las migraciones
phpartisanmigrate: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:

```
phpartisanmigrate:refresh
```

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

```
phpartisanmigrate:status
```

SchemaBuilder

Una vez creada una migración tenemos que completar sus métodos up y down para indicar la tabla que queremos crear o el campo que queremos modificar. En el método down siempre tendremos que añadir la operación inversa, eliminar la tabla que se ha creado en el método up o eliminar la columna que se ha añadido. Esto nos permitirá deshacer migraciones dejando la base de datos en el mismo estado en el que se encontraban antes de que se añadieran.

Para especificar la tabla a crear o modificar, así como las columnas y tipos de datos de las mismas, se utiliza la clase *Schema*. Esta clase tiene una serie de métodos que nos permitirá especificar la estructura de las tablas independientemente del sistema de base de datos que utilicemos.

Crear y borrar una tabla

Para añadir una nueva tabla a la base de datos se utiliza el siguiente constructor:

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

Donde el primer argumento es el nombre de la tabla y el segundo es una función que recibe como parámetro un objeto del tipo*Blueprint*que utilizaremos para configurar las columnas de la tabla.

En la sección`down`de la migración tendremos que eliminar la tabla que hemos creado, para esto usaremos alguno de los siguientes métodos:

```
Schema::drop('users');

Schema::dropIfExists('users');
```

Al crear una migración con el comando de Artisan`make:migration` ya nos viene este código añadido por defecto, la creación y eliminación de la tabla que se ha indicado y además se añaden un par de columnas por defecto (*id* y *timestamps*).

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();
});
```

Schema define muchos tipos de datos que podemos utilizar para definir las columnas de una tabla, algunos de los principales son:

Comando	Tipo de campo
$table->boolean('confirmed');	BOOLEAN
$table->enum('choices', array('foo', 'bar'));	ENUM
$table->float('amount');	FLOAT
$table->increments('id');	Clave principal tipo INTEGER con Auto-Increment
$table->integer('votes');	INTEGER
$table->mediumInteger('numbers');	MEDIUMINT
$table->smallInteger('votes');	SMALLINT
$table->tinyInteger('numbers');	TINYINT
$table->string('email');	VARCHAR
$table->string('name', 100);	VARCHAR con la longitud indicada
$table->text('description');	TEXT
$table->timestamp('added_on');	TIMESTAMP

Comando	Tipo de campo
$table->timestamps();	Añadelos*timestamps* "created_at" y "updated_at"
->nullable()	Indicar que la columna permite valores NULL
->default($value)	Declare a default value for a column
->unsigned()	Añade UNSIGNED a las columnas tipo INTEGER

Los tres últimos se pueden combinar con el resto de tipos para crear, por ejemplo, una columna que permita nulos, con un valor por defecto y de tipo *unsigned*.

Para consultar todos los tipos de datos que podemos utilizar podéis consultar la documentación de Laravel en:

http://laravel.com/docs/5.1/migrations#creating-columns

Añadir índices

Schema soporta los siguientes tipos de índices:

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

Claves ajenas

Con *Schema* también podemos definir claves ajenas entre tablas:

```
$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`".

Nota: 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 "*ondelete*" y "*onupdate*":

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

Para eliminar una clave ajena, en el método `down` de la migración tenemos que utilizar el siguiente código:

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

Para indicar la clave ajena a eliminar tenemos que seguir el siguiente patrón para especificar el nombre

```
<tabla>_<columna>_foreign.
```

Donde "tabla" es el nombre de la tabla actual y "columna" el nombre de la columna sobre la que se creo la clave ajena.

Inicialización de la base de datos (DatabaseSeeding)

Laravel también facilita la inserción de datos iniciales o datos *semilla* en la base de datos. Esta opción es muy útil para tener datos de prueba cuando estamos desarrollando una web o para crear tablas que ya tienen que contener una serie de datos en producción.

Los ficheros de "semillas" se encuentran en la carpeta `database/seeds`. Por defecto Laravel incluye el fichero `DatabaseSeeder` con el siguiente contenido:

```
<?php

useIlluminate\Database\Seeder;
useIlluminate\Database\Eloquent\Model;

classDatabaseSeederextendsSeeder
{
/**
     * Run the database seeds.
     * @return void
```

```
     */
publicfunctionrun()
{
//...
    }
}
```

Al lanzar la incialización se llamará por defecto al método `run` de la clase `DatabaseSeeder`. Desde aquí podemos crear las semillas de varias formas:

1. Escribir el código para insertar los datos dentro del propio método `run`
2. Crear otros métodos dentro de la clase `DatabaseSeeder`y llamarlos desde el método `run`. De esta forma podemos separar mejor las inicializaciones.
3. Crear otros ficheros *Seeder*y llamarlos desde el método `run` es la clase principal.

Según lo que vayamos a hacer nos puede interesar una opción u otra. Por ejemplo, si el código que vamos a escribir es poco nos puede sobrar con las opciones 1 o 2, sin embargo si vamos a trabajar bastante con las inicializaciones quizás lo mejor es la opción 3.

A continuación se incluye un ejemplo de la opción 1:

```
classDatabaseSeederextendsSeeder
{
publicfunctionrun()
{
// Borramos los datos de la tabla
        DB::table('users')->delete();

// Añadimos una entrada a esta tabla
User::create(array('email' =>'foo@bar.com'));
}
}
```

Como se puede ver en el ejemplo en general tendremos que eliminar primero los datos de la tabla en cuestión y posteriormente añadir los datos. Para insertar datos en una tabla podemos utilizar el método que se usa en el ejemplo o alguna de las otras opciones que se verán en las siguientes secciones sobre "Constructor de consultas" y "Eloquent ORM".

Crear ficheros semilla

Como hemos visto en el apartado anterior, podemos crear más ficheros o clases *semilla* para modularizar mejor el código de las inicializaciones. De esta forma podemos crear un fichero de semillas para cada una de las tablas o

modelos de datos que tengamos.

En la carpeta `database/sedes` podemos añadir más ficheros PHP con clases que extiendan de `Seeder`para definir nuestros propios ficheros de "semillas". El nombre de los ficheros suele seguir el mismo patrón`<nombre-tabla>TableSeeder`, por ejemplo "`UsersTableSeeder`". Artisan incluye un comando que nos facilitará crear los ficheros de semillas y que además incluirán las estructura base de la clase. Por ejemplo, para crear el fichero de inicialización de la tabla de usuarios haríamos:

```
phpartisanmake:seederUsersTableSeeder
```

Para que esta nueva clase se ejecute tenemos que llamarla desde el método `run` de la clase principal `DatabaseSeeder` de la forma:

```
classDatabaseSeederextendsSeeder
{
publicfunctionrun()
{
        Model::unguard();

$this->call(UsersTableSeeder::class);

Model::reguard();
    }
}
```

El método `call` lo que hace es llamar al método `run` de la clase indicada. Además en el ejemplo hemos añadido las llamadas a `unguard` y a `reguard`, que lo que hacen es desactivar y volver a activar (respectivamente) la inserción de datos masiva o por lotes.

Ejecutar la inicialización de datos

Una vez definidos los ficheros de semillas, cuando queramos ejecutarlos para rellenar de datos la base de datos tendremos que usar el siguiente comando de Artisan:

```
phpartisandb:seed
```

Constructor de consultas (QueryBuilder)

Laravel incluye una serie de clases que nos facilita la construcción de consultas y otro tipo de operaciones con la base de datos. Además, al utilizar estas clases, creamos una notación mucho más legible, compatible con todos los

tipos de bases de datos soportados por Laravel y que nos previene de cometer errores o de ataques por inyección de código SQL.

Consultas

Para realizar una "Select" que devuelva todas las filas de una tabla utilizaremos el siguiente código:

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

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

En el ejemplo se utiliza el constructor`DB::tabla`indicando el nombre de la tabla sobre la que se va a realizar la consulta, y por último se llama al método`get()`para obtener todas las filas de la misma.

Si queremos obtener un solo elemento podemos utilizar`first`en lugar de`get`, de la forma:

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

echo$user->name;
```

Clausula *where*

Para filtrar los datos usamos la clausula`where`, indicando el nombre de la columna y el valor a filtrar:

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

echo$user->name;
```

En este ejemplo, la clausula`where` filtrará todas las filas cuya columna `name` sea igual a `Pedro`. Si queremos realizar otro tipo de filtrados, como columnas que tengan un valor mayor (>), mayor o igual (>=), menor (<), menor o igual (<=), distinto del indicado (<>) o usar el operador `like`, lo podemos indicar como segundo parámetro de la forma:

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

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

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

Si añadimos más clausulas `where` a la consulta por defecto se unirán mediante el operador lógico `AND`. En caso de que queramos utilizar el operador lógico `OR` lo tendremos que realizar usando `orWhere` de la forma:

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

orderBy / groupBy / having_

También podemos utilizar los métodos `orderBy`, `groupBy` y `having` en las consultas:

```
$users = DB::table('users')
                    ->orderBy('name', 'desc')
                    ->groupBy('count')
                    ->having('count', '>', 100)
->get();
```

Offset / Limit

Si queremos indicar un *offset* o *limit* lo realizaremos mediante los métodos `skip` (para el *offset*) y `take` (para *limit*), por ejemplo:

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Transacciones

Laravel también permite crear transacciones sobre un conjunto de operaciones:

```
DB::transaction(function()
{
    DB::table('users')->update(array('votes' =>1));

DB::table('posts')->delete();
});
```

En caso de que se produzca cualquier excepción en las operaciones que se realizan en la transacción se desharían todos los cambios aplicados hasta ese momento de forma automática.

Más informacion

Para más información sobre la construcción de *Querys* (*join, insert, update, delete*, agregados, etc.) podéis consultar la documentación de Laravel en su sitio web:

http://laravel.com/docs/5.1/queries

Modelos de datos mediante ORM

El mapeado objeto-relacional (más conocido por su nombre en inglés, *Object-Relationalmapping*, 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.

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 espaciode nombres.

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

```
<?php
namespaceApp;

useIlluminate\Database\Eloquent\Model;

classUserextendsModel
{
//...
}
```

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

```
phpartisanmake:modelUser
```

Este comando creará el fichero `User.php` dentro de la carpeta `app` con el código básico de un modelo que hemos visto en el ejemplo anterior.

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
namespaceApp;

useIlluminate\Database\Eloquent\Model;

classUserextendsModel
{
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';`.

Nota: 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;`.

A continuación se muestra un ejemplo de un modelo de *Eloquent* en el que se añaden todas las especificaciones que hemos visto:

```
<?php
namespaceApp;

useIlluminate\Database\Eloquent\Model;

classUserextendsModel
{
protected$table = 'my_users';
protected$primaryKey = 'my_id'
public$timestamps = false;
}
```

Uso de un modelo de datos

Una vez creado el modelo ya podemos empezar a utilizarlo para recuperar datos de la base de datos, para insertar nuevos datos o para actualizarlos. El sitio correcto donde realizar estas acciones es en el controlador, el cual se los tendrá que pasar a la vista ya preparados para su visualización.

Es importante que para su utilización indiquemos al inicio de la clase el espacio de nombres del modelo o modelos a utilizar. Por ejemplo, si vamos a usar los modelos`User` y `Orders`tendríamos que añadir:

```
useApp\User;
useApp\Orders;
```

Consultar datos

Para obtener todas las filas de la tabla asociada a un modelo usaremos el método `all()`:

```
$users = User::all();

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

Este método nos devolverá un array de resultados, donde cada item del array será una instancia del modelo `User`. Gracias a esto al obtener un elemento del array podemos acceder a los campos o columnas de la tabla como si fueran propiedades del objeto (`$user->name`).

Nota: Todos los métodos que se describen en la sección de "Constructor de consultas" y en la documentación de Laravel sobre "QueryBuilder" también se pueden utilizar en los modelos Eloquent. Por lo tanto podremos utilizar *where, orWhere, first, get, orderBy, groupBy, having, skip, take,* etc. para elaborar las consultas.

Eloquent también incorpora el método `find($id)` para buscar un elemento a partir del identificador único del modelo, por ejemplo:

```
$user = User::find(1);
echo$user->name;
```

Si queremos que se lance una excepción cuando no se encuentre un modelo podemos utilizar los métodos `findOrFail` o `firstOrFail`. Esto nos permite capturar las excepciones y mostrar un error 404 cuando sucedan.

```
$model = User::findOrFail(1);

$model = User::where('votes', '>', 100)->firstOrFail();
```

A continuación se incluyen otros ejemplos de consultas usando Eloquent con algunos de los métodos que ya habíamos visto en la sección "Constructor de consultas":

```
// Obtener 10 usuarios con más de 100 votos
$users = User::where('votes', '>', 100)->take(10)->get();

// Obtener el primer usuario con más de 100 votos
$user = User::where('votes', '>', 100)->first();
```

También podemos utilizar los métodos agregados para calcular el total de registros obtenidos, o el máximo, mínimo, media o suma de una determinada columna. Por ejemplo:

```
$count = User::where('votes', '>', 100)->count();
$price = Orders::max('price');
$price = Orders::min('price');
$price = Orders::avg('price');
$total = User::sum('votes');
```

Insertar datos

Para añadir una entrada en la tabla de la base de datos asociada con un modelo simplemente tenemos que crear una nueva instancia de dicho modelo, asignar los valores que queramos y por último guardarlos con el método `save()`:

```
$user = new User;
$user->name = 'Juan';
$user->save();
```

Para obtener el identificador asignado en la base de datos después de guardar (cuando se trate de tablas con índice auto-incremental), lo podremos recuperar simplemente accediendo al campo `id` del objeto que habíamos creado, por ejemplo:

```
$insertedId = $user->id;
```

Actualizar datos

Para actualizar una instancia de un modelo es muy sencillo, solo tendremos que recuperar en primer lugar la instancia que queremos actualizar, a continuación modificarla y por último guardar los datos:

```
$user = User::find(1);
$user->email = 'juan@gmail.com';
$user->save();
```

Borrar datos

Para borrar una instancia de un modelo en la base de datos simplemente tenemos que usar su método `delete()`:

```
$user = User::find(1);
$user->delete();
```

Si por ejemplo queremos borrar un conjunto de resultados también podemos usar el método `delete()` de la forma:

```
$affectedRows = User::where('votes', '>', 100)->delete();
```

Más información

Para más información sobre cómo crear relaciones entre modelos, *eagerloading*, etc. podéis consultar directamente la documentación de Laravel en:

http://laravel.com/docs/5.1/eloquent