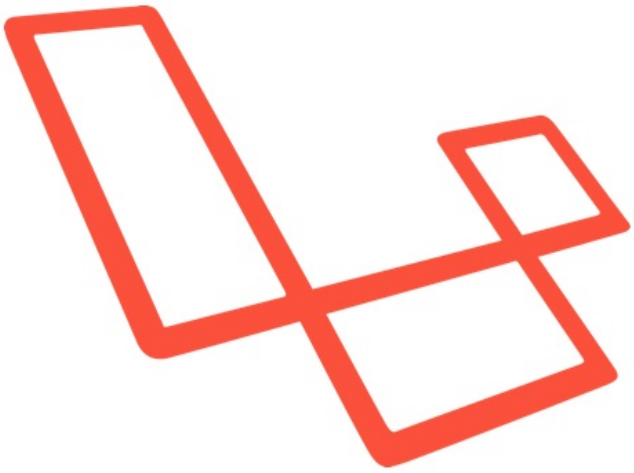


Laravel 5

The PHP Framework For Web Artisans

Introducción · Conceptos básicos · Ejemplos · Artisan · Rutas
Controladores · Vistas · Plantillas · Blade · Middleware · Modelos
Bases de datos · Migraciones · Eloquent ORM · Assets · Ficheros
Formularios · Validación · Gestión de Usuarios · Autenticación · API
RESTful · Paquetes y extensiones · Respuestas especiales · JSON

Antonio Javier Gallego Sánchez

Tabla de contenido

Contenidos

En este libro se ve una introducción al *framework* de desarrollo Web Laravel. Se parte de los conceptos más básicos: introducción, instalación, estructura de un proyecto, ejemplos sencillos de uso, etc. Pero se llegan a ver aspectos más avanzados como es el uso de base de datos, el control de usuarios o la creación de una API.

Para la lectura de este manual es necesario tener conocimientos sobre HTML, CSS, Javascript y PHP, ya que todos ellos se dan por sabidos.

A continuación se muestra un detalle del contenido de los capítulos en los que se divide el libro:

Introducción

¿Qué es Laravel?

Laravel es un *framework* de código abierto para el desarrollo de aplicaciones web en PHP 5 que posee una sintaxis simple, expresiva y elegante. Fue creado en 2011 por Taylor Otwell, inspirándose en Ruby on Rails y Symfony, de los cuales ha adoptado sus principales ventajas.

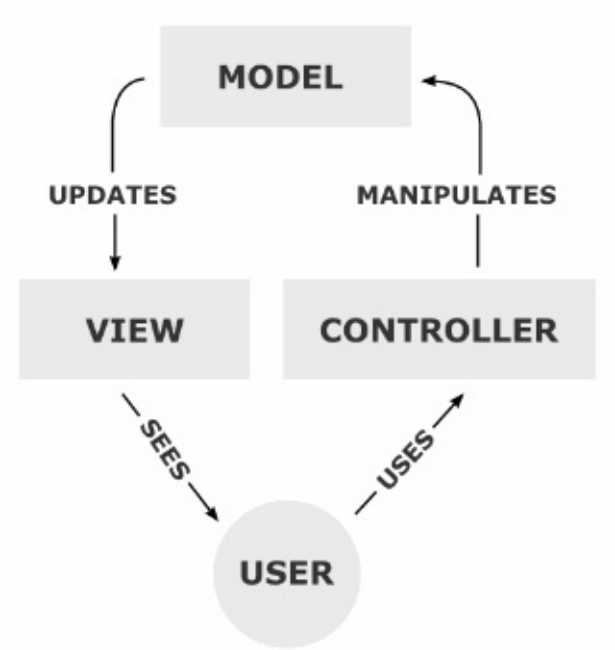
Laravel facilita el desarrollo simplificando el trabajo con tareas comunes como la autenticación, el enrutamiendo, gestión sesiones, el almacenamiento en caché, etc. Algunas de las principales características y ventajas de Laravel son:

- Esta diseñado para desarrollar bajo el patrón MVC (modelo - vista - controlador), centrándose en la correcta separación y modularización del código. Lo que facilita el trabajo en equipo, así como la claridad, el mantenimiento y la reutilización del código.
- Integra un sistema ORM de mapeado de datos relacional llamado Eloquent aunque también permite la construcción de consultas directas a base de datos mediante su *Query Builder*.
- Permite la gestión de bases de datos y la manipulación de tablas desde código, manteniendo un control de versiones de las mismas mediante su sistema de *Migraciones*.
- Utiliza un sistema de plantillas para las vistas llamado Blade, el cual hace uso de la cache para darle mayor velocidad. Blade facilita la creación de vistas mediante el uso de *layouts*, herencia y secciones.
- Facilita la extensión de funcionalidad mediante paquetes o librerías externas. De esta forma es muy sencillo añadir paquetes que nos faciliten el desarrollo de una aplicación y nos ahorren mucho tiempo de programación.
- Incorpora un intérprete de línea de comandos llamado *Artisan* que nos ayudará con un montón de tareas rutinarias como la creación de distintos componentes de código, trabajo con la base de datos y migraciones, gestión de rutas, cachés, colas, tareas

programadas, etc.

MVC: Modelo - Vista - Controlador

El modelo–vista–controlador (MVC) es un patrón de arquitectura de software que separa los datos y la lógica de negocio de una aplicación de la interfaz de usuario y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario. Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento.



De manera genérica, los componentes de MVC se podrían definir como sigue:

- El Modelo: Es la representación de la información con la cual el sistema opera, por lo tanto gestiona todos los accesos a dicha información, tanto consultas como actualizaciones. Las peticiones de acceso o manipulación de información llegan al 'modelo' a través del 'controlador'.
- El Controlador: Responde a eventos (usualmente acciones del usuario) e invoca peticiones al 'modelo' cuando se hace alguna solicitud de información (por ejemplo, editar un documento o un registro en una base de datos). Por tanto se podría decir que el 'controlador' hace de intermediario entre la 'vista' y el 'modelo'.

- La Vista: Presenta el 'modelo' y los datos preparados por el controlador al usuario de forma visual. El usuario podrá interactuar con la vista y realizar otras peticiones que se enviarán al controlador.

Novedades en la versión 5

En Laravel 5 se han incluido un montón de novedades y cambios con respecto a la versión anterior. Si vienes de la versión 4 y quieres actualizar tus proyectos a la nueva versión puedes seguir la guía de actualización de su página: http://laravel.com/docs/5.1/upgrade. Según el proyecto, en ocasiones será más sencillo empezar desde cero y trasladar el código que modificar el proyecto anterior. Si seguimos la guía podremos actualizar nuestros proyectos pero es recomendable que antes revises toda la nueva documentación ya que hay muchas nuevas funcionalidades que podemos aprovechar.

Algunas de las novedades que se incluyen en Laravel 5 son:

- Han cambiado completamente la estructura de carpetas, todo o casi todo ha cambiado de sitio. La nueva estructura está pensada para separar o modularizar mejor nuestro código y para agrupar mejor las clases de código relacionado.
- Se ha incluido el espacio de nombres para organizar y cargar el código. En la versión anterior todo estaba en el espacio de nombres global. En esta nueva versión se recomienda separar nuestro código bajo distintos espacios de nombres según su funcionalidad y después requerirlos en las clases que lo utilicen.
- Los filtros, que antes estaban todos mezclados en un único fichero, se han pasado a un nuevo tipo de clase llamada "*Middleware*", la cual incluye también nuevas funcionalidades.
- Mediante un comando de Artisan es posible cachear las rutas de nuestra aplicación, esto, según la documentación, llega a acelerar hasta 100x la carga.
- En Artisan también se han añadido otros métodos que nos facilitarán la generación de código como controladores, modelos, etc.
- Inyección de dependencias en controladores, vistas y otros elementos, lo cual nos creará un código más limpio, modular, fácil de mantener, y con un bajo acoplamiento entre sus componentes.
- Han mejorado el sistema para la gestión y autenticación de usuarios, incluyendo más funcionalidades como el throttling, OAuth o políticas de autorización.
- Han incluido mejoras en el sistema de colas, permitiendo definirlos como si fueran comandos de Artisan y después, de una forma muy sencilla, crear tareas repetitivas o programadas sobre ellos.
- El sistema de entornos de configuración también ha cambiado completamente. Ahora en lugar de usar carpetas anidadas para cada entorno se utiliza el sistema "*DotEnv*" a

partir de un único fichero con la configuración de cada usuario o entorno centralizada.
- Se han incluido paquetes para facilitarnos la gestión de suscripciones y pagos (Cashier), para trabajar con los assets (Elixir) y para la conexión mediante OAuth con servicios como Facebook, Twitter, Google o GitHub (mediante Socialite).

Además, en la nueva versión de Laravel se han adoptado dos nuevos estándares: PSR-4 (http://www.php-fig.org/psr/psr-4/) para la carga automática de clases a partir de su ruta de archivos, y PSR-2 (http://www.php-fig.org/psr/psr-2/) como guía de estilo del código fuente.

Capítulo 1.
Primeros pasos

En este primer capítulo vamos a dar los primeros pasos en el mundo Laravel. Aprenderemos desde como instalar un servidor Web, como crear nuestro primer proyecto y configurarlo, y llegaremos a ver algunas de las funcionalidades principales de este *framework* para ser capaces de crear nuestra primera Web.

Instalación de Laravel

Para la utilización de Laravel en primer lugar necesitamos tener instalado un servidor Web con PHP `>=` 5.5.9, MySQL y la extensión MCrypt de PHP. Una vez instalado el servidor procederemos a instalar la utilidad Composer y por último la librería de Laravel. A continuación se describen los pasos a seguir.

Instalación del servidor Web XAMPP

Como servidor Web para Mac vamos a utilizar la versión de XAMPP de ApacheFriends. Para su instalación seguiremos los siguientes pasos:

- En primer lugar abrimos su página web "https://www.apachefriends.org", entramos en la sección de descargas y bajamos la última versión para Mac.
- Esto descargará una imagen tipo DMG (en el caso de Mac), hacemos doble clic encima para iniciar el proceso de instalación.
- Al finalizar habrá instalado el servidor en la ruta `/Applications/XAMPP`.

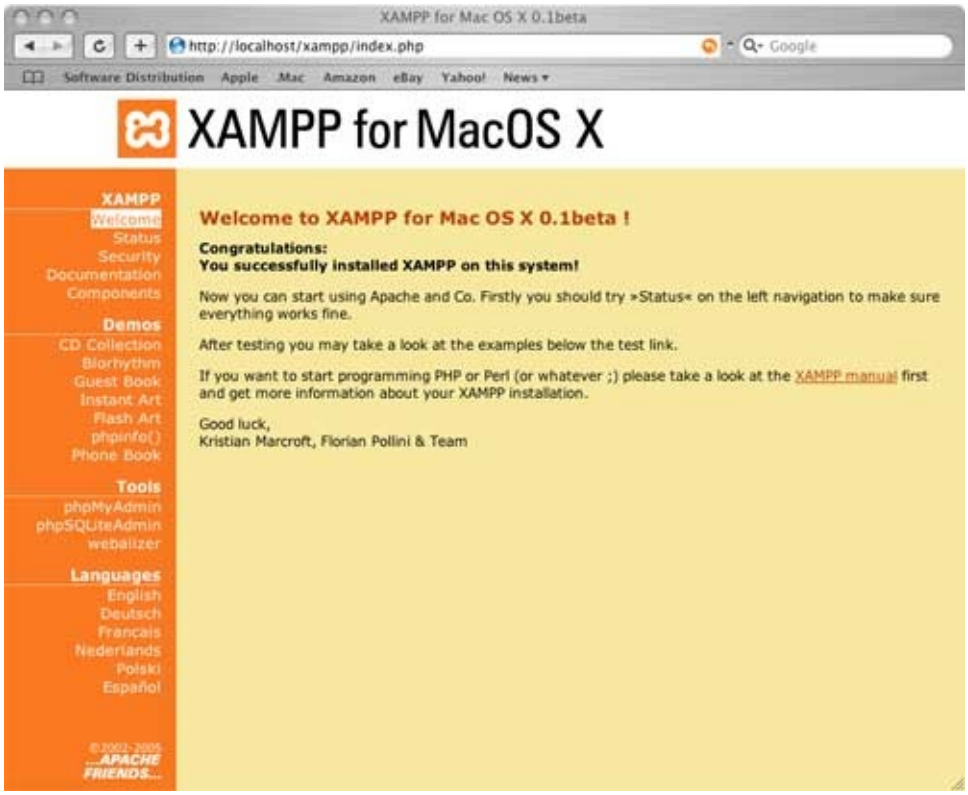
Con esto ya tenemos un servidor Web instalado en nuestro ordenador. Ahora para iniciarlo y pararlo solo tendremos que acceder a su Panel de Control e iniciar o parar Apache y MySQL. El nombre de la aplicación de Panel de Control de XAMPP es "**manager-osx**".

Desde este mismo panel de control, además de poder iniciar o parar los servicios, podemos ver el log, abrir la carpeta de disco donde tenemos que almacenar nuestro proyectos Web (por defecto situada en `/Applications/XAMPP/htdocs`) o abrir la URL de nuestro servidor web en un navegador (http://localhost/xampp/).

Para comprobar que el servidor se ha instalado correctamente podemos abrir la siguiente URL en el navegador:

```
http://localhost
```

Esto nos mostrará la página por defecto de XAMPP, que contiene algunos links para comprobar el estado del software instalado y algunos ejemplos de programación.



> Si ya hubiese un servidor web instalado en el ordenador es posible que entre en conflicto con XAMPP y no permita iniciar el servicio de Apache. En este caso tendremos que detener el otro servidor (`sudo /usr/sbin/apachectl stop`) para poder utilizar XAMPP.

Desde la versión 4 de Laravel, la creación de un proyecto nuevo se realiza con Composer. Veamos entonces que es Composer y que necesitamos para usarlo.

Instalación de Composer

Composer es un gestor de dependencias para PHP. Esto quiere decir que permite descargar de sus repositorios todas las librerías y las dependencias con las versiones requeridas que el proyecto necesite.

Instalar Composer es muy sencillo por linea de comandos. Si accedemos a su página web en "https://getcomposer.org/" podemos consultar las instrucciones, simplemente tendremos que hacer:

```
$ curl -sS https://getcomposer.org/installer | php
$ sudo mv composer.phar /usr/local/bin/composer
```

El primer comando descarga el archivo `composer.phar` en nuestro ordenador (`.phar` es una extensión para aplicaciones PHP comprimidas). El segundo comando mueve el archivo descargado a la carpeta `bin` para que Composer pueda ser ejecutado de forma global.

> En Windows tendremos que descargar el instalador que encontraremos en la Web de Composer y ejecutarlo para instalar la aplicación.

Por último verificamos la instalación con el siguiente comando:

```
$ composer
```

Si la instalación se ha realizado correctamente se nos debería mostrar una lista de los comandos y opciones que tiene Composer.

Instalar Laravel mediante Composer

En la carpeta raíz de nuestro servidor web (`/Applications/XAMPP/htdocs`) ejecutamos el siguiente comando:

```
$ composer create-project laravel/laravel miweb --prefer-dist
```

Esto nos descargará la última versión de Laravel y creará una carpeta llamada `miweb` con todo el contenido ya preparado. Si nos apareciera algún error de permisos tendríamos que ejecutar de nuevo el mismo comando pero con `sudo` .

Si accedemos a la carpeta que se acaba de crear (`cd miweb`) y ejecutamos `$ php artisan` comprobaremos que nos aparece el siguiente error:

```
Mcrypt PHP extension required.
```

Esto es debido a que Laravel requiere la extensión Mcrypt para su utilización. En la siguiente sección se explica como solucionar este error.

Instalación de Mcrypt

Para instalar Mcrypt seguiremos los siguientes pasos:

- Ejecutamos `which php` para comprobar la ruta del php utilizado, la cual deberá ser (`/usr/bin/php`).
- A continuación escribimos:

```
sudo nano ~/.bash_profile
```

- Al final de este fichero añadimos la siguiente línea:

```
PATH="/Applications/XAMPP/xamppfiles/bin:$PATH"
```

- Y por último presionamos Ctrl+O para guardar los cambios y Ctrl-X para salir.

Con esto hemos añadido al PATH del sistema la ruta de los ejecutables que incorpora XAMPP, entre los cuales se incluye Mcrypt. Para que se actualice el PATH cerramos el terminal actual y volvemos a abrir uno. Ahora si escribimos otra vez `which php` nos tendrá que aparecer la nueva ruta: `/Applications/XAMPP/xamppfiles/bin/php`. Para comprobar que ya funciona el CLI de Laravel accedemos a la ruta donde lo hemos instalado (`/Applications/XAMPP/htdocs/miweb`) y ejecutamos `php artisan`, lo cual nos tendría que mostrar:

```
Laravel Framework version 5.1.23 (LTS)

Usage:
  command [options] [arguments]

Options:
  -h, --help            Display this help message
  ...
  ...
```

Con esto ya tenemos instalado un servidor Web y Laravel funcionando, pero todavía nos falta terminar de configurar nuestra nueva Web con Laravel para que funcione correctamente.

Configuración inicial de Laravel

Por último solo nos queda revisar la configuración y permisos de nuestro nuevo proyecto con Laravel.

La configuración (a partir de Laravel 5) se encuentra almacenada en ficheros dentro de la carpeta " `config` ". Para empezar no es necesario modificar ninguna configuración. Sin embargo, cada uno de los archivos de configuración está bien documentado, por lo que puedes revisarlos libremente por si quieres cambiar algún valor (más adelante volveremos sobre este tema).

La mayoría de las opciones de configuración tienen su valor puesto directamente en el fichero, pero hay algunas que se cargan a través de variables de entorno utilizando el sistema *DotEnv*. Estas variables tienen que estar definidas en el fichero `.env` de la raíz de la aplicación, y mediante el método `env('NOMBRE', 'VALOR-POR-DEFECTO')` se cargan y se asignan a una opción de configuración. Esto permite separar configuración según el entorno

o el usuario que lo utilice simplemente cambiando el fichero `.env`. Así por ejemplo podemos tener un fichero `.env` para el entorno de desarrollo local, otro para producción, otro para pruebas, etc.

El único valor que nos tenemos que asegurar de que esté correctamente configurado es la `key` o clave de la aplicación. Esta clave es una cadena de 32 caracteres que se utiliza para codificar los datos. En caso de no establecerla (revisar el fichero `config/app.php` y la variable `APP_KEY` definida en el fichero `.env`) nuestra aplicación no será segura. Para crearla simplemente tenemos que ejecutar el siguiente comando en la carpeta raíz de nuestra aplicación:

```
php artisan key:generate
```

Además tenemos que establecer los permisos de algunas carpetas especiales. En general no es necesario añadir permisos de escritura para los archivos de nuestra aplicación, solo tendremos que hacerlo para las carpetas `storage` y `bootstrap/cache`, que es donde Laravel almacena los logs, sesiones, chachés, etc. Para establecer estos permisos simplemente tenemos que ejecutar:

```
sudo chmod -R 777 storage
sudo chmod -R 777 bootstrap/cache
```

> En Windows no será necesario dar permisos de escritura a estas carpetas.

Comprobación de Laravel

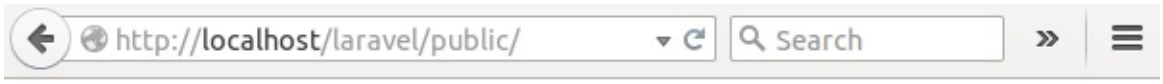
Una vez completados todos los pasos anteriores ya podemos comprobar que nuestra nueva página web con Laravel funciona correctamente. Para esto abrimos un navegador y accedemos a la siguiente URL:

```
http://localhost/<nombre-de-tu-proyecto-web>/public/
```

Que en nuestro caso sería:

```
http://localhost/miweb/public/
```

Esto nos tendría que mostrar una página web por defecto como la siguiente:



Laravel 5

Funcionamiento básico

En esta sección vamos a analizar la estructura de un proyecto, es decir, donde va cada cosa, y a continuación veremos el ciclo de vida de una petición en Laravel.

Estructura de un proyecto

Al crear un nuevo proyecto de Laravel se nos generará una estructura de carpetas y ficheros para organizar nuestro código. Es importante que conozcamos para que vale cada elemento y donde tenemos que colocar nuestro código. En este manual lo iremos viendo poco a poco, por lo que más adelante se volverán a explicar algunos de estos elementos más en detalle. Pero de momento vamos a explicar brevemente las carpetas que más utilizaremos y las que mejor tendremos que conocer:

- `app` – Contiene el código principal de la aplicación. Esta carpeta a su vez está dividida en muchas subcarpetas que analizaremos en la siguiente sección.
- `config` – Aquí se encuentran todos los archivos de configuración de la aplicación: base datos, cache, correos, sesiones o cualquier otra configuración general de la aplicación.
- `database` – En esta carpeta se incluye todo lo relacionado con la **definición de la base de datos** de nuestro proyecto. Dentro de ella podemos encontrar a su vez tres carpetas: *factores*, *migrations* y *seeds*. En el capítulo sobre base de datos analizaremos mejor su contenido.
- `public` – Es la única carpeta pública, la única que debería ser **visible** en nuestro servidor web. 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` – Esta carpeta contiene a su vez tres carpetas: *assets*, *views* y *lang*:
    - `resources/views` – Este directorio contiene las vistas de nuestra aplicación. En general serán plantillas de HTML que usan los controladores para mostrar la información. Hay que tener en cuenta que en esta carpeta no se almacenan los Javascript, CSS o imágenes, ese tipo de archivos se tienen que guardar en la carpeta `public`.

    - `resources/lang` – En esta carpeta se guardan archivos PHP que contienen arrays con los textos de nuestro sitio web en diferentes lenguajes, solo será necesario utilizarla en caso que se desee que la aplicación se pueda traducir.
    - `resources/assets` – Se utiliza para almacenar los fuentes de los assets tipo *less* o *sass* que se tendrían que compilar para generar las hojas de estilo públicas. No es necesario usar esta carpeta ya que podemos escribir directamente las las hojas de estilo dentro de la carpeta *public*.

- `bootstrap` – En esta carpeta se incluye el código que se carga para procesar cada una de las llamadas a nuestro proyecto. Normalmente no tendremos que modificar nada de esta carpeta.
- `storage` – En esta carpeta Laravel almacena toda la información interna necesarios para la ejecución de la web, como son los archivos de sesión, la caché, la compilación de las vistas, meta información y los logs del sistema. Normalmente tampoco tendremos que tocar nada dentro de esta carpeta, unicamente se suele acceder a ella para consultar los logs.
- `tests` – Esta carpeta se utiliza para los ficheros con las pruebas automatizadas. Laravel incluye un sistema que facilita todo el proceso de pruebas con PHPUnit.
- `vendor` – En esta carpeta se alojan todas las librerías y dependencias que conforman el *framework* de Laravel. Esta carpeta tampoco la tendremos que modificar, ya que todo el código que contiene son librerías que se instalan y actualizan mediante la herramienta Composer.

Además en la carpeta raíz también podemos encontrar dos ficheros muy importantes y que también utilizaremos:

- `.env` – Este fichero ya lo hemos mencionado en la sección de instalación, se utiliza para almacenar los valores de configuración que son propios de la máquina o instalación actual. Lo que nos permite cambiar fácilmente la configuración según la máquina en la que se instale y tener opciones distintas para producción, para distintos desarrolladores, etc. Importante, este fichero debería estar en el `.gitignore`.
- `composer.json` – Este fichero es el utilizado por Composer para realizar la instalación de Laravel. En una instalación inicial únicamente se especificará la instalación de un paquete, el propio *framework* de Laravel, pero podemos especificar la instalación de otras librerías o paquetes externos que añadan funcionalidad a Laravel.

Carpeta *App*

La carpeta *app* es la que contiene el código principal del proyecto, como son las rutas, controladores, filtros y modelos de datos. Si accedemos a esta carpeta veremos que contiene a su vez muchas sub-carpetas, pero la principal que vamos a utilizar es la carpeta `Http`:

- `app/Http/Controllers` – Contiene todos los archivos con las clases de los controladores que sirven para interactuar con los modelos, las vistas y manejar la lógica de la aplicación.
- `app/Http/Middleware` – Son los filtros o clases intermedias que podemos utilizar para realizar determinadas acciones, como la validación de permisos, antes o después de la ejecución de una petición a una ruta de nuestro proyecto web.
- `app/Http/routes.php` – Este documento define todas las rutas de nuestro sitio web, enlazando una URL del navegador con un método de un controlador. Además nos permite realizar validaciones (mediante *Middleware*) y otras operaciones sobre las rutas de nuestro sitio.

Además de esta carpeta encontraremos muchas otras como *Console*, *Events*, *Exceptions*, *Jobs*, *Listeners*, *Policies* y *Providers*. Más adelante veremos algunas de estas carpetas pero de momento la única que vamos a utilizar es *Http*.

En la raíz de *app* también podemos encontrar el fichero `User.php`. Este fichero es un modelo de datos que viene predefinido por Laravel para trabajar con los usuarios de la web, que incluye métodos para hacer login, registro, etc. En el capítulo sobre bases de datos hablaremos más sobre esto.

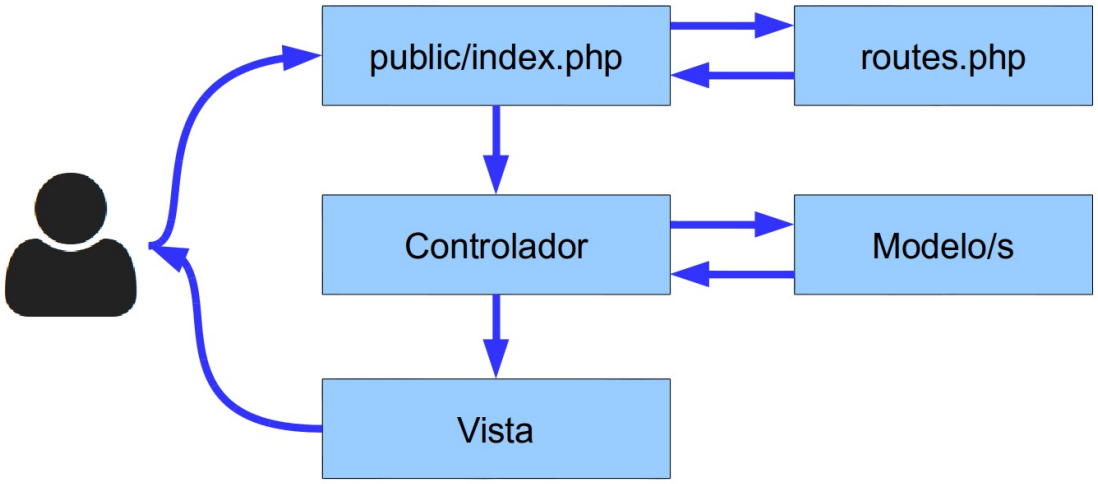
Funcionamiento básico

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.

A continuación se incluye un pequeño esquema de este funcionamiento:



En las siguientes secciones iremos viendo cada uno de estos apartados por separado. En primer lugar se estudiará como podemos definir las rutas que tiene nuestra aplicación y como las tenemos que enlazar con los controladores. Seguidamente se verán los controladores y vistas, dejando los modelos de datos y el uso de la base de datos para más adelante.

Rutas

Las rutas de nuestra aplicación aplicación se tienen que definir en el fichero `app/Http/routes.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. Empezaremos viendo el primer tipo de rutas y en secciones posteriores se tratará como enlazarlas con una vista o con 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 `routes.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`).

De la misma forma podemos definir rutas para peticiones tipo PUT o DELETE:

```
Route::put('foo/bar', function () {
    //
});

Route::delete('foo/bar', function () {
    //
});
```

Si queremos que una ruta se defina a la vez para varios verbos 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]+'))
```

Generar una ruta

Cuando queramos generar la URL hasta una ruta podemos utilizar el siguiente método:

```
$url = url('foo');
```

Con este método nos aseguraremos que la URL sea válida y además se le añadirá el dominio que tengamos definido en los ficheros de configuración. En general no será necesaria su utilización y simplemente podremos escribir la ruta a mano hasta una dirección de la forma: `/foo` (anteponiendo la barra `/` para asegurarnos que la ruta sea a partir de la raíz del dominio de nuestro sitio). Sin embargo se recomienda la utilización de este método en general para evitar problemas de generación de rutas no existentes o relativas (si se nos olvidase anteponer la `/`).

Checkpoint

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 mas 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.

Para ver una lista de todas las opciones que incluye Artisan podemos ejecutar el siguiente comando en un 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
```

En secciones anteriores ya hemos utilizado uno de estos comandos, `php artisan key:generate`, para generar la clave de encriptación de nuestro proyecto Web. Poco a poco iremos viendo más opciones de Artisan, de momento vamos a comentar solo dos opciones importantes: el listado de rutas y la generación de código.

Listado de rutas

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.

Generación de código

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 --plain
```

En las siguientes secciones utilizaremos algunos de estos conceptos y también veremos más comandos de Artisan.

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.

Definir vistas

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>
```

Referenciar y devolver vistas

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' => 'Javi'));
});
```

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 "Javi".

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');
});
```

Pasar datos a una vista

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));
});
```

Laravel además ofrece una alternativa que crea una notación un poco más clara. En lugar de pasar un array como segundo parámetro podemos utilizar el método `with` para indicar una a una las variables o contenidos que queremos enviar a la vista:

```
$view = view('home')->with('nombre', 'Javi');

$view = view('user.profile')
            ->with('user', $user)
            ->with('editable', false);
```

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.

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 !!}.
```

> Nota: En general siempre tendremos que usar las llaves dobles, en especial si vamos a mostrar datos que son proporcionados por los usuarios de la aplicación. Esto evitará que inyecten símbolos que produzcan errores o inyecten codigo javascript que se ejecute sin que nosotros queramos. Por lo tanto, este último método solo tenemos que utilizarlo si estamos seguros de que no queremos que se escape el contenido.

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 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
```

Esta son las estructuras de control más utilizadas. Ademas de estas *Blade* define algunas más que podemos ver directamente en su documentación: http://laravel.com/docs/5.1/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 de 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 condenido 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')
```

Ejercicios

En los ejercicios de esta sección del curso vamos a desarrollar una pequeña web para la gestión interna de un videoclub, empezaremos por definir las rutas y vistas del sitio y poco a poco en los siguientes ejercicios la iremos completando hasta terminar el sitio web completo.

El objetivo es realizar un sitio web para la gestión interna en un videoclub, el cual estará protegido mediante usuario y contraseña. Una vez autorizado el acceso, el usuario podrá listar el catálogo de películas, ver información detallada de una película, realizar búsquedas o filtrados y algunas operaciones más de gestión.

Ejercicio 1 - Instalación de Laravel (0.5 puntos)

En primer lugar tenemos que instalar todo lo necesario para poder realizar el sitio web con Laravel. Para esto seguiremos las explicaciones del apartado "Instalación de Laravel" que hemos visto en la teoría para instalar un servidor Web y Composer.

Una vez instalado crearemos un nuevo proyecto de Laravel en la carpeta `videoclub`, lo configuraremos (clave de seguridad, permisos, etc.) y probaremos que todo funcione correctamente.

Ejercicio 2 - Definición de las rutas (0.5 puntos)

En este ejercicio vamos a definir las rutas principales que va a tener nuestro sitio web. Para empezar simplemente indicaremos que las rutas devuelvan una cadena (así podremos comprobar que se han creado correctamente). A continuación se incluye una tabla con las rutas a definir (todas de tipo GET) y el texto que tienen que mostrar:

Ruta	Texto a mostrar
/	Pantalla principal
auth/login	Login usuario
auth/logout	Logout usuario
catalog	Listado películas
catalog/show/{id}	Vista detalle película {id}
catalog/create	Añadir película
catalog/edit/{id}	Modificar película {id}

Para comprobar que las rutas se hayan creado correctamente utiliza el comando de `artisan` que devuelve un listado de rutas y además prueba también las rutas en el navegador.

Ejercicio 3 - *Layout* principal de las vistas con Bootstrap (1 punto)

En este ejercicio vamos a crear el *layout* base que van a utilizar el resto de vistas del sitio web y además incluiremos la librería Bootstrap para utilizarla como estilo base.

En primer lugar accedemos a la web "http://getbootstrap.com/" y descargamos la librería. Esto nos bajará un fichero zip comprimido con tres carpetas (*js*, *css* y *fonts*) que tenemos que extraer en la carpeta " `public/assets/bootstrap` " (tendremos que crear las carpetas " `/assets/bootstrap` ").

También nos tenemos que descargar desde los materiales de los ejercicios la plantilla para la barra de navegación principal (`navbar.blade.php`) y la almacenamos en la carpeta `resources/views/partials` .

A continuación vamos a crear el *layout* principal de nuestro sitio:

- Creamos el fichero `resources/views/layouts/master.blade.php` .
- Le añadimos como contenido la plantilla base HTML que propone Bootstrap en su documentación "http://getbootstrap.com/getting-started/#template", modificando los siguientes elementos:
    - Cambiamos las rutas para la carga de los assets (*css* y *js*) que hemos almacenado en local. Para generar la ruta completa y que encuentre los recursos tendremos que escribir los siguientes comandos:

```
{{ url('/assets/bootstrap/css/bootstrap.min.css') }}
{{ url('/assets/bootstrap/js/bootstrap.min.js') }}
```

- Dentro de la sección `<body>` del HTML, eliminamos el texto que viene de ejemplo (`<h1>Hello, world!</h1>`) e incluimos la barra de navegación que hemos guardado antes utilizando el siguiente código:

```
@include('partials.navbar')
```

- A continuación de la barra de navegación añadimos la sección principal donde aparecerá el contenido de la web:

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

Con esto ya hemos definido el layout principal, sin embargo todavía no podemos probarlo ya que no está asociado a ninguna ruta. En el siguiente ejercicio realizaremos los cambios necesarios para poder verlo y además añadiremos el resto de vistas hijas.

Ejercicio 4 - Crear el resto de vistas (1 punto)

En este ejercicio vamos terminar una primera versión estable de la web. En primer lugar crearemos las vistas asociadas a cada ruta, las cuales tendrán que extender del *layout* que hemos hecho en el ejercicio anterior y mostrar (en la sección de `content` del layout) el texto de ejemplo que habíamos definido para cada ruta en el ejercicio 2. En general todas las vistas tendrán un código similar al siguiente (variando únicamente la sección `content`):

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

@section('content')

    Pantalla principal

@stop
```

Para organizar mejor las vistas las vamos a agrupar en sub-carpetas dentro de la carpeta `resources/views` siguiendo la siguiente estructura:

Vista	Carpeta	Ruta asociada
`home.blade.php`	`resources/views/`	/
`login.blade.php`	`resources/views/auth/`	auth/login
`index.blade.php`	`resources/views/catalog/`	catalog
`show.blade.php`	`resources/views/catalog/`	catalog/show/{id}
`create.blade.php`	`resources/views/catalog/`	catalog/create
`edit.blade.php`	`resources/views/catalog/`	catalog/edit/{id}

> Creamos una vista separada para todas las rutas excepto para la ruta "logout", la cual no tendrá ninguna vista.

Por último vamos a actualizar las rutas del fichero `routes.php` para que se carguen las vistas que acabamos de crear. Acordaros que para referenciar las vistas que están dentro de carpetas la barra `/` de separación se transforma en un punto, y que además, como segundo parámetro, podemos pasar datos a la vista. A continuación se incluyen algunos ejemplos:

```
return view('home');
return view('catalog.index');
return view('catalog.show', array('id'=>$id));
```

Una vez hechos estos cambios ya podemos probarlo en el navegador, el cual debería mostrar en todos los casos la plantilla base con la barra de navegación principal y los estilos de Bootstrap aplicados. En la sección principal de contenido de momento solo podremos ver los textos que hemos puesto de ejemplo.

Capítulo 2.
Controladores, filtros y formularios

En este capítulo empezaremos a utilizar realmente el patrón MVC y veremos como añadir controladores a nuestros proyectos web. Además también veremos el concepto de filtros mediante *Middleware* aplicados sobre el fichero de rutas y como realizar redirecciones en Laravel. Por último se incluye una sección de ejercicios para practicar con todo lo aprendido.

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
<?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 --plain
```

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
```

> 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
<?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:

```php
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 como 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
<?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:

```php
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 como 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 este 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 codificacón.

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>
```

Ejercicios

En los ejercicios de esta parte vamos a continuar con el sitio Web que empezamos para la gestión de un videoclub. Primero añadiremos los controladores y métodos asociados a cada ruta, y posteriormente también completaremos las vistas usando formularios y el sistema de plantillas *Blade*.

Ejercicio 1 - Controladores (1 punto)

En este primer ejercicio vamos a crear los controladores necesarios para gestionar nuestra aplicación y además actualizaremos el fichero de rutas para que los utilice.

Empezamos por añadir los dos controladores que nos van a hacer falta: `CatalogController.php` y `HomeController.php`. Para esto tenéis que utilizar el comando de Artisan que permite crear un controlador vacío (sin métodos).

A continuación vamos a añadir los métodos de estos controladores. En la siguiente tabla resumen podemos ver un listado de los métodos por controlador y las rutas que tendrán asociadas:

Ruta	Controlador	Método
/	HomeController	getHome
catalog	CatalogController	getIndex
catalog/show/{id}	CatalogController	getShow
catalog/create	CatalogController	getCreate
catalog/edit/{id}	CatalogController	getEdit

Acordaros que los métodos `getShow` y `getEdit` tendrán que recibir como parámetro el `$id` del elemento a mostrar o editar, por lo que la definición del método en el controlador tendrá que ser como la siguiente:

```
public function getShow($id)
{
    return view('catalog.show', array('id'=>$id));
}
```

Por último vamos a cambiar el fichero de rutas `routes.php` para que todas las rutas que teníamos definidas (excepto las de *login* y *logout* que las dejaremos como están) apunten a los nuevos métodos de los controladores, por ejemplo:

```
Route::get('/', 'HomeController@getHome');
```

El código que teníamos puesto para cada ruta con el `return` con la generación de la vista lo tenéis que mover al método del controlador correspondiente.

Ejercicio 2 - Completar las vistas (2 puntos)

En este ejercicio vamos a terminar los métodos de los controladores que hemos creado en el ejercicio anterior y además completaremos las vistas asociadas:

Método HomeController@getHome

En este método de momento solo vamos a hacer una redirección a la acción que muestra el listado de películas del catálogo: `return redirect()->action('CatalogController@getIndex');` . Más adelante tendremos que comprobar si el usuario está logueado o no, y en caso de que no lo este redirigirle al formulario de login.

Método CatalogController@getIndex

Este método tiene que mostrar un listado de todas las películas que tiene el videoclub. El listado de películas lo podéis obtener del fichero `array_peliculas.php` facilitado con los materiales. Este array de películas lo tenéis que copiar como variable miembro de la clase (más adelante las almacenaremos en la base de datos). En el método del controlador simplemente tendremos que modificar la generación de la vista para pasarle este array de películas completo (`$this->arrayPeliculas`).

Y en la vista correspondiente simplemente tendremos que incluir el siguiente trozo de código en su sección `content` :

```
<div class="row">

    @foreach( $arrayPeliculas as $key => $pelicula )
    <div class="col-xs-6 col-sm-4 col-md-3 text-center">

        <a href="{{ url('/catalog/show/' . $key ) }}">
            <img src="{{$pelicula['poster']}}" style="height:200px"/>
            <h4 style="min-height:45px;margin:5px 0 10px 0">
                {{$pelicula['title']}}
            </h4>
        </a>

    </div>
    @endforeach

</div>
```

Como se puede ver en el código, en primer lugar se crea una fila (usando el sistema de rejilla de Bootstrap) y a continuación se realiza un bucle *foreach* utilizando la notación de *Blade* para iterar por todas las películas. Para cada película obtenemos su posición en el array y sus datos asociados, y generamos una columna para mostrarlos. Es importante que nos fijemos en como se itera por los elementos de un array de datos y en la forma de acceder a los valores. Además se ha incluido un enlace para que al pulsar sobre una película nos lleve a la dirección `/catalog/show/{$key}`, siendo `key` la posición de esa película en el array.

Método CatalogController@getShow

Este método se utiliza para mostrar la vista detalle de una película. Hemos de tener en cuenta que el método correspondiente recibe un identificador que (de momento) se refiere a la posición de la película en el array. Por lo tanto, tendremos que coger dicha película del array (`$this->arrayPeliculas[$id]`) y pasársela a la vista.

En esta vista vamos a crear dos columnas, la primera columna para mostrar la imagen de la película y la segunda para incluir todos los detalles. A continuación se incluye la estructura HTML que tendría que tener esta pantalla:

```
<div class="row">

    <div class="col-sm-4">

        {{-- TODO: Imagen de la película --}}

    </div>
    <div class="col-sm-8">

        {{-- TODO: Datos de la película --}}

    </div>
</div>
```

En la columna de la izquierda completamos el TODO para insertar la imagen de la película. En la columna de la derecha se tendrán que mostrar todos los datos de la película: título, año, director, resumen y su estado. Para mostrar el estado de la película consultaremos el valor `rented` del array, el cual podrá tener dos casos:

- En caso de estar disponible (false) aparecerá el estado "Película disponible" y un botón azul para "Alquilar película".
- En caso de estar alquilada (true) aparecerá el estado "Película actualmente alquilada" y un botón rojo para "Devolver película".

Además tenemos que incluir dos botones más, un botón que nos llevará a editar la película y otro para volver al listado de películas.

> Nota: los botones de alquilar/devolver de momento no tienen que funcionar. Acordaros que en Bootstrap podemos transformar un enlace en un botón, simplemente aplicando las clases "btn btn-default" (más info en: http://getbootstrap.com/css/#buttons).

Esta pantalla finalmente tendría que tener una apariencia similar a la siguiente:

Método CatalogController@getCreate

Este método devuelve la vista " `catalog.create` " para añadir una nueva película. Para crear este formulario en la vista correspondiente nos podemos basar en el contenido de la plantilla " `catalog_create.php` ". Esta plantilla tiene una serie de TODOs que hay que completar. En total tendrá que tener los siguientes campos:

Label	Name	Tipo de campo
Título	title	text
Año	year	text
Director	director	text
Poster	poster	text
Resumen	synopsis	textarea

Además tendrá un botón al final con el texto "Añadir película".

> De momento el formulario no funcionará. Más adelante lo terminaremos.

Método CatalogController@getEdit

Este método permitirá modificar el contenido de una película. El formulario será exactamente igual al de añadir película, así que lo podemos copiar y pegar en esta vista y simplemente cambiar los siguientes puntos:

- El título por "Modificar película".
- El texto del botón de envío por "Modificar película".
- Añadir justo debajo de la apertura del formulario el campo oculto para indicar que se va a enviar por PUT. Recordad que Laravel incluye el método {{method_field('PUT')}} que

nos ayudará a hacer esto.

De momento no tendremos que hacer nada más, más adelante lo completaremos para que se rellene con los datos de la película a editar.

Capítulo 3.
Base 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 cual 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 como 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. En la siguiente sección veremos en detalle que es esto, de momento solo decir que 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:

```
php artisan migrate:install
```

> 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 como 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 *Schema builder* (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:migration create_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:migration add_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
<?php

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

class CreateUsersTable extends Migration
{
    /**
     * Run the migrations.
     * @return void
     */
    public function up()
    {
        //
    }

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

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 como especificar esto) tenemos que lanzar la migración con el siguiente comando:

```
php artisan migrate
```

> Si nos aparece el error "class not found" lo podremos solucionar llamando a `composer dump-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:

```
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
```

Schema Builder

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
$table->timestamps();	Añade los *timestamps* "created_at" y "updated_at"
->nullable()	Indicar que la columna permite valores NULL
->default($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` ".

> 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');
```

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 (*Database Seeding*)

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

use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

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

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:

```
class DatabaseSeeder extends Seeder
{
    public function run()
    {
        // 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/seeds` 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:

```
php artisan make:seeder UsersTableSeeder
```

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

```
class DatabaseSeeder extends Seeder
{
    public function run()
    {
        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:

```
php artisan db:seed
```

Constructor de consultas (*Query Builder*)

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 ($users as $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-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.

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`:

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    //...
}
```

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

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

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
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;` .

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

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    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:

```
use App\User;
use App\Orders;
```

Consultar datos

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

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

foreach( $users as $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 "Query Builder" 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 como crear relaciones entre modelos, *eager loading*, etc. podéis consultar directamente la documentación de Laravel en:

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

Ejercicios

En estos ejercicios vamos a continuar con el proyecto del videoclub que habíamos empezado en sesiones anteriores y le añadiremos todo lo referente a la gestión de la base de datos.

Ejercicio 1 - Configuración de la base de datos y migraciones (1 punto)

En primer lugar vamos a configurar correctamente la base de datos. Para esto tenemos que actualizar los ficheros `config/database.php` y `.env` para indicar que vamos a usar una base de datos tipo MySQL llamada " `videoclub` " junto con el nombre de usuario y contraseña de acceso.

> Nota: XAMPP por defecto crea el usuario de base de datos *root* sin contraseña.

A continuación abrimos *PHPMyAdmin* y creamos la nueva base de datos llamada `videoclub`. Para comprobar que todo se ha configurado correctamente vamos a un terminal en la carpeta de nuestro proyecto y ejecutamos el comando que crea la tabla de migraciones. Si todo va bien podremos actualizar desde *PHPMyAdmin* y comprobar que se ha creado esta tabla dentro de nuestra nueva base de datos.

> Si nos diese algún error tendremos que revisar los valores indicados en el fichero `.env`. En caso de ser correctos es posible que también tengamos que reiniciar el servidor o terminal que tengamos abierto.

Ahora vamos a crear la tabla que utilizaremos para almacenar el catálogo de películas. Ejecuta el comando de Artisan para crear la migración llamada `create_movies_table` para la tabla `movies`. Una vez creado edita este fichero para añadir todos los campos necesarios, estos son:

Campo	Tipo	Valor por defecto
id	Autoincremental	
title	String	
year	String de longitud 8	
director	String de longitud 64	
poster	String	
rented	Booleano	false
synopsis	Text	
timestamps	Timestamps de Eloquent	

> Recuerda que en el método `down` de la migración tienes que deshacer los cambios que has hecho en el método `up`, en este caso sería eliminar la tabla.

Por último ejecutaremos el comando de Artisan que añade las nuevas migraciones y comprobaremos en *PHPMyAdmin* que la tabla se ha creado correctamente con los campos que le hemos indicado.

Ejercicio 2 - Modelo de datos (0.5 puntos)

En este ejercicio vamos a crear el modelo de datos asociado con la tabla *movies*. Para esto usaremos el comando apropiado de Artisan para crear el modelo llamado `Movie`.

Una vez creado este fichero lo abriremos y comprobaremos que el nombre de la clase sea el correcto y que herede de la clase `Model`. Y ya está, no es necesario hacer nada más, el cuerpo de la clase puede estar vacío (`{}`), todo lo demás se hace automáticamente!

Ejercicio 3 - Semillas (1 punto)

Ahora vamos a proceder a rellenar la tabla de la base de datos con los datos iniciales. Para esto editamos el fichero de semillas situado en `database/seeds/DatabaseSeeder.php` y seguiremos los siguientes pasos:

- Creamos un método privado (dentro de la misma clase) llamado `seedCatalog()` que se tendrá que llamar desde el método `run` de la forma:

```
public function run()
{
  self::seedCatalog();
  $this->command->info('Tabla catálogo inicializada con datos!');
}
```

- Movemos el array de películas que se facilitaba en los materiales y que habíamos copiado dentro del controlador `CatalogController` a la clase de semillas (`DatabaseSeeder.php`), guardándolo de la misma forma, como variable privada de la clase.
- Dentro del nuevo método `seedCatalog()` realizamos las siguientes acciones:
  - En primer lugar borramos el contenido de la tabla `movies` con `DB::table('movies')->delete();` .
  - Y a continuación añadimos el siguiente código:

```
foreach( $this->arrayPeliculas as $pelicula ) {
    $p = new Movie;
    $p->title = $pelicula['title'];
    $p->year = $pelicula['year'];
    $p->director = $pelicula['director'];
    $p->poster = $pelicula['poster'];
    $p->rented = $pelicula['rented'];
    $p->synopsis = $pelicula['synopsis'];
    $p->save();
}
```

Por último tendremos que ejecutar el comando de Artisan que procesa las semillas y una vez realizado abriremos *PHPMyAdmin* para comprobar que se rellenado la tabla *movies* con el listado de películas.

> Si te aparece el error "*Fatal error: Class 'Movie' not found*" revisa si has indicado el espacio de nombres del modelo que vas a utilizar (`use App\Movie;`).

Ejercicio 4 - Uso de la base de datos (1 punto)

En este último ejercicio vamos a actualizar los métodos del controlador `CatalogController` para que obtengan los datos desde la base de datos. Seguiremos los siguientes pasos:

- Modificar el método `getIndex` para que obtenga toda la lista de películas desde la base de datos usando el modelo `Movie` y que se la pase a la vista.

- Modificar el método `getShow` para que obtenga la película pasada por parámetro usando el método `findOrFail` y se la pase a la vista.
- Modificar el método `getEdit` para que obtenga la película pasada por parámetro usando el método `findOrFail` y se la pase a la vista.

> Si al probarlo te aparece el error "*Class 'App\Http\Controllers\Movie' not found*" revisa si has indicado el espacio de nombres del modelo que vas a utilizar (`use App\Movie;`).

Ya no necesitaremos más el array de películas (`$arrayPeliculas`) que habíamos puesto en el controlador, así que lo podemos comentar o eliminar.

Ahora tendremos que actualizar las vistas para que en lugar de acceder a los datos del array los obtenga del **objeto** con la película. Para esto cambiaremos en todos los sitios donde hayamos puesto `$pelicula['campo']` por `$pelicula->campo` .

Además, en la vista `catalog/index.blade.php` , en vez de utilizar el índice del array (`$key`) como identificador para crear el enlace a `catalog/show/{id}` , tendremos que utilizar el campo `id` de la película (`$pelicula->id`). Lo mismo en la vista `catalog/show.blade.php` , para generar el enlace de editar película tendremos que añadir el identificador de la película a la ruta `catalog/edit` .

Capítulo 4.
Datos de entrada y Control de usuarios

En este cuarto capítulo vamos a aprender como recoger los datos de entrada de formularios o de algún otro tipo de petición (como por ejemplo una petición de una API). También veremos como leer ficheros de entrada.

En la sección de control de usuarios se tratará todo lo referente a la gestión de los usuarios de una aplicación web, desde como crear la tabla de usuarios, como registrarlos, autenticarlos en la aplicación, cerrar la sesión o como proteger las partes privadas de nuestra aplicación de accessos no permitidos.

Datos de entrada

Laravel facilita el acceso a los datos de entrada del usuario a través de solo unos pocos métodos. No importa el tipo de petición que se haya realizado (POST, GET, PUT, DELETE), si los datos son de un formulario o si se han añadido a la *query string*, en todos los casos se obtendrán de la misma forma.

Para conseguir acceso a estos métodos Laravel utiliza inyección de dependencias. Esto es simplemente añadir la clase `Request` al constructor o método del controlador en el que lo necesitemos. Laravel se encargará de inyectar dicha dependencia ya inicializada y directamente podremos usar el parámetro para obtener los datos de entrada. A continuación se incluye un ejemplo:

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

use Illuminate\Http\Request;
use Illuminate\Routing\Controller;

class UserController extends Controller
{
    public function store(Request $request)
    {
        $name = $request->input('name');

        //...
    }
}
```

En este ejemplo como se puede ver se ha añadido la clase `Request` como parámetro al método `store`. Laravel automáticamente se encarga de inyectar estas dependencias por lo que directamente podemos usar la variable `$request` para obtener los datos de entrada.

Si el método del controlador tuviera más parámetros simplemente los tendremos que añadir a continuación de las dependencias, por ejemplo:

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

A continuación veremos los métodos y datos que podemos obtener a partir de la variable `$request`.

Obtener los valores de entrada

Para obtener el valor de una variable de entrada usamos el método `input` indicando el nombre de la variable:

```
$name = $request->input('name');
```

También podemos especificar un valor por defecto como segundo parámetro:

```
$name = $request->input('name', 'Pedro');
```

Comprobar si una variable existe

Si lo necesitamos podemos comprobar si un determinado valor existe en los datos de entrada:

```
if ($request->has('name'))
{
    //...
}
```

Obtener datos agrupados

O también podemos obtener todos los datos de entrada a la vez (en un array) o solo algunos de ellos:

```
// Obtener todos:
$input = $request->all();

// Obtener solo los campos indicados:
$input = $request->only('username', 'password');

// Obtener todos excepto los indicados:
$input = $request->except('credit_card');
```

Obtener datos de un array

Si la entrada proviene de un *input* tipo array de un formulario (por ejemplo una lista de *checkbox*), si queremos podremos utilizar la siguiente notación con puntos para acceder a los elementos del array de entrada:

```
$input = $request->input('products.0.name');
```

JSON

Si la entrada está codificada formato JSON (por ejemplo cuando nos comunicamos a través de una API es bastante común) también podremos acceder a los diferentes campos de los datos de entrada de forma normal (con los métodos que hemos visto, por ejemplo: `$name = $request->input('name');`).

Ficheros de entrada

Laravel facilita una serie de clases para trabajar con los ficheros de entrada. Por ejemplo para obtener un fichero que se ha enviado en el campo con nombre `photo` y guardarlo en una variable, tenemos que hacer:

```
$file = $request->file('photo');
```

Si queremos podemos comprobar si un determinado campo tiene un fichero asignado:

```
if ($request->hasFile('photo')) {
{
    //...
}
```

El objeto que recuperamos con `$request->file()` es una instancia de la clase `Symfony\Component\HttpFoundation\File\UploadedFile` , la cual extiende la clase de PHP `SplFileInfo` (http://php.net/manual/es/class.splfileinfo.php), por lo tanto, tendremos muchos métodos que podemos utilizar para obtener datos del fichero o para gestionarlo.

Por ejemplo, para comprobar si el fichero que se ha subido es válido:

```
if ($request->file('photo')->isValid())
{
    //...
}
```

O para mover el fichero de entrada a una ruta determinada:

```
// Mover el fichero a la ruta conservando el nombre original:
$request->file('photo')->move($destinationPath);

// Mover el fichero a la ruta con un nuevo nombre:
$request->file('photo')->move($destinationPath, $fileName);
```

Otros métodos que podemos utilizar para recuperar información del fichero son:

```
// Obtener la ruta:
$path = $request->file('photo')->getRealPath();

// Obtener el nombre original:
$name = $request->file('photo')->getClientOriginalName();

// Obtener la extensión:
$extension = $request->file('photo')->getClientOriginalExtension();

// Obtener el tamaño:
$size = $request->file('photo')->getSize();

// Obtener el MIME Type:
$mime = $request->file('photo')->getMimeType();
```

Control de usuarios

Laravel incluye una serie de métodos y clases que harán que la implementación del control de usuarios sea muy rápida y sencilla. De hecho, casi todo el trabajo ya está hecho, solo tendremos que indicar donde queremos utilizarlo y algunos pequeños detalles de configuración.

Por defecto, al crear un nuevo proyecto de Laravel, ya se incluye todo lo necesario:

- La configuración predeterminada en `config/auth.php`.
- La migración para la base de datos de la tabla de usuarios con todos los campos necesarios.
- El modelo de datos de usuario (`User.php`) dentro de la carpeta `app` con toda la implementación necesaria.
- Los controladores para gestionar todas las acciones relacionadas con el control de usuarios (dentro de `App\Http\Controllers\Auth`).

En el siguiente apartado veremos la configuración inicial del sistema de autenticación y los módulos por los que está compuesto. Más adelante revisaremos también como utilizar este sistema para proteger nuestro sitio web.

Configuración inicial

La **configuración** del sistema de autenticación se puede encontrar en el fichero `config/auth.php`, el cual contiene varias opciones (bien documentadas) que nos permitirán, por ejemplo: cambiar el sistema de autenticación (que por defecto es a través de Eloquent), cambiar el modelo de datos usado para los usuarios (por defecto será `User`) y cambiar la tabla de usuarios (que por defecto será `users`). Si vamos a utilizar estos valores no será necesario que realicemos ningún cambio.

La **migración** de la tabla de usuarios (llamada `users`) también está incluida (ver carpeta `database/migrations`). Por defecto incluye todos los campos necesarios (ver el código siguiente), pero si necesitamos alguno más lo podemos añadir y guardar por ejemplo la dirección o el teléfono del usuario. A continuación se incluye el código de la función `up` de la migración:

```
Schema::create('users', function (Blueprint $table) {
    $table->increments('id');
    $table->string('name');
    $table->string('email')->unique();
    $table->string('password', 60);
    $table->rememberToken();
    $table->timestamps();
});
```

Como se puede ver el nombre de la tabla es `users`, con un índice `id` autoincremental, y los campos de `name`, `email`, `password`. El campo *email* es único y el *password* tiene una longitud de 60 caracteres. Además se añaden los *timestamps* que usa Eloquent automáticamente y el `remember_token` para recordar la sesión del usuario.

En la carpeta `app` se encuentra el **modelo de datos** (llamado `User.php`) para trabajar con los usuarios. Esta clase ya incluye toda la implementación necesaria y por defecto no tendremos que modificar nada. Pero si queremos podemos modificar esta clase para añadirle más métodos o relaciones con otras tablas, etc.

Laravel también incluye dos **controladores** (`AuthController` y `PasswordController`) para la autenticación de usuarios, ambos los puedes encontrar en el espacio de nombres `App\Http\Controllers\Auth` (y en la misma carpeta). `AuthController` incluye métodos para ayudarnos en el proceso de autenticación, registro y cierre de sesión; mientras que `PasswordController` contiene la lógica para ayudarnos en el proceso de restaurar una contraseña. Para la mayoría de aplicaciones con estos métodos será suficiente y no tendremos que añadir nada más.

Lo único que falta por añadir y configurar correctamente para que todo funcione son las rutas y las vistas. A continuación veremos estos puntos.

Rutas

Por defecto Laravel no incluye ninguna ruta para el control de usuarios (ya que muchas aplicaciones no requerirán de estas o querrán poner las suyas propias). Si lo deseamos podemos añadir las siguientes rutas al fichero `app/Http/routes.php` para gestionar las acciones de acceso de usuarios (*login*), cerrar sesión (*logout*) y registro:

```
// Rutas para autenticación...
Route::get('auth/login', 'Auth\AuthController@getLogin');
Route::post('auth/login', 'Auth\AuthController@postLogin');
Route::get('auth/logout', 'Auth\AuthController@getLogout');

// Rutas para registro...
Route::get('auth/register', 'Auth\AuthController@getRegister');
Route::post('auth/register', 'Auth\AuthController@postRegister');
```

Al acceder a la ruta `auth/login` por GET se mostrará el formulario de login, y lo mismo para la ruta `auth/register` para el registro. Las rutas `auth/login` y `auth/register` por POST se encargarán de procesar los datos enviados por los formularios. Y al acceder a la ruta `auth/logout` se cerrará la sesión.

Por último lo que nos falta por definir son los formularios o vistas que se mostrarán al acceder a estas rutas.

Vistas

Aunque los métodos de los controladores ya están definidos, Laravel no proporciona el contenido de las vistas a mostrar, por lo tanto las tendremos que añadir nosotros mismos. Estas vistas se tienen que almacenar en la carpeta `resources/views/auth` con los nombres `login.blade.php` para el formulario de *login* y `register.blade.php` para el formulario de registro. Estos nombres y rutas son obligatorios ya que los controladores que incluye Laravel accederán a ellos.

A continuación se incluye el código de un formulario de ejemplo para la vista de *login*:

```
<!-- resources/views/auth/login.blade.php -->

<form method="POST" action="/auth/login">
    {!! csrf_field() !!}

    <div>
        <label for="email">Email</label>
        <input type="email" name="email" id="email" value="{{ old('email') }}">
    </div>

    <div>
        <label for="password">Contraseña</label>
        <input type="password" name="password" id="password">
    </div>

    <div>
        <input type="checkbox" name="remember"> Recuérdame
    </div>

    <div>
        <button type="submit">Entrar</button>
    </div>
</form>
```

A continuación se incluye el código de un formulario de ejemplo para la vista de registro:

```
<!-- resources/views/auth/register.blade.php -->

<form method="POST" action="/auth/register">
    {!! csrf_field() !!}

    <div>
        <label for="name">Nombre</label>
        <input type="text" name="name" id="name" value="{{ old('name') }}">
    </div>

    <div>
        <label for="email">Email</label>
        <input type="email" name="email" id="email" value="{{ old('email') }}">
    </div>

    <div>
        <label for="password">Contraseña</label>
        <input type="password" name="password" id="password">
    </div>

    <div>
        <label for="password_confirmation">Confirmar contraseña</label>
        <input type="password" name="password_confirmation" id="password_confirmation">
    </div>

    <div>
        <button type="submit">Registrar</button>
    </div>
</form>
```

Autenticación de un usuario

Una vez configurado todo el sistema, añadidas las rutas y las vistas para realizar el control de usuarios ya podemos utilizarlo. Si accedemos a la ruta `auth/login` nos aparecerá la vista con el formulario de login, solicitando nuestro email y contraseña para acceder. Si además añadimos un campo tipo *checkbox* llamado "*remember*" nos permitirá indicar si deseamos que la sesión permanezca abierta hasta que se cierre manualmente. Es decir, aunque se cierre el navegador y pasen varios días el usuario seguiría estando autorizado.

Si los datos introducidos son correctos se creará la sesión del usuario y se le redirigirá a la ruta " `/home` ". Si queremos cambiar esta ruta tenemos que definir la propiedad `redirectPath` en el controlador `AuthController` , por ejemplo:

```
protected $redirectPath = '/dashboard';
```

Si los datos del usuario no son correctos se le volverá a redirigir a la ruta con el formulario de *login* (`/auth/login`). Si por alguna razón hubiéramos cambiado dicha ruta lo tendremos que indicar también definiendo la propiedad `loginPath` en el controlador `AuthController` por ejemplo:

```
protected $loginPath = '/login';
```

Registro de un usuario

Si accedemos a la ruta `auth/register` nos aparecerá la vista con el formulario de registro, solicitándonos los campos nombre, *email* y contraseña. Al pulsar el botón de envío del formulario se llamará a la ruta `auth/register` por POST y se almacenará el nuevo usuario en la base de datos.

Si no hemos añadido ningún campo más en la migración no tendremos que configurar nada más. Sin embargo si hemos añadido algún campo más a la tabla de usuarios tendremos que actualizar dos métodos del controlador `AuthController` . En el método `validator` simplemente tendremos que añadir dicho campo al array de validaciones (y solo en el caso que necesitemos validarlo). Y en el método `create` tendremos que añadir los campos adicionales que deseemos almacenar. El código de este método es el siguiente:

```
protected function create(array $data)
{
    return User::create([
        'name' => $data['name'],
        'email' => $data['email'],
        'password' => bcrypt($data['password']),
    ]);
}
```

Como podemos ver utiliza el modelo de datos `User` para crear el usuario y almacenar las variables que recibe en el array de datos `$data` . En este array de datos nos llegarán todos los valores de los campos del formulario, por lo tanto, si añadimos más campos al formulario y a la tabla de usuarios simplemente tendremos que añadirlos también en este método.

Es importante destacar que la contraseña se cifra usando el método `bcrypt` . Por lo tanto las contraseñas se almacenaran cifradas en la base de datos.

Acceder a los datos del usuario atenticado

Una vez que el usuario está autenticado podemos acceder a los datos del mismo a través del método `Auth::user()`, por ejemplo:

```
user = Auth::user();
```

Este método nos devolverá `null` en caso de que no esté autenticado. Si estamos seguros de que el usuario está autenticado (porque estamos en una ruta protegida) podremos acceder directamente a sus propiedades:

```
$email = Auth::user()->email;
```

> Importante: para utilizar la clase `Auth` tenemos que añadir el espacio de nombres `use Auth;`, de otra forma nos aparecerá un error indicando que no puede encontrar la clase.

El usuario también se inyecta en los parámetros de entrada de la petición (la clase `Request`). Por lo tanto, si en un método de un controlador usamos la inyección de dependencias también podremos acceder a los datos del usuario:

```
class ProfileController extends Controller
{
    public function updateProfile(Request $request)
    {
        if ($request->user()) {
            $email = $request->user()->email;
        }
    }
}
```

Cerrar la sesión

Si accedemos a la ruta `auth/logout` que hemos definido antes se cerrará la sesión y se redirigirá a la ruta de `home`. Todo esto lo hará automáticamente el método `getLogout` del controlador `AuthController`.

Para cerrar manualmente la sesión del usuario actualmente autenticado tenemos que utilizar el método:

```
Auth::logout();
```

Posteriormente podremos hacer una redirección a una página principal para usuarios no autenticados.

> Importante: para utilizar la clase `Auth` tenemos que añadir el espacio de nombres `use Auth;`, de otra forma nos aparecerá un error indicando que no puede encontrar la clase.

Comprobar si un usuario está autenticado

Para comprobar si el usuario actual se ha autenticado en la aplicación podemos utilizar el método `Auth::check()` de la forma:

```
if (Auth::check())
{
    // El usuario está correctamente autenticado
}
```

Sin embargo, lo recomendable es utilizar *Middleware* (como veremos a continuación) para realizar esta comprobación antes de permitir el acceso a determinadas rutas.

> Importante: para utilizar la clase `Auth` tenemos que añadir el espacio de nombres `use Auth;`, de otra forma nos aparecerá un error indicando que no puede encontrar la clase.

Proteger rutas

El sistema de autenticación de Laravel también incorpora una serie de filtros o *Middleware* (ver carpeta `app/Http/Middleware`) para comprobar que el usuario que accede a una determinada ruta o grupo de rutas esté autenticado. En concreto para proteger el acceso a rutas y solo permitir su visualización por usuarios correctamente autenticados usaremos el *middleware* `app\Http\Middleware\Authenticate.php` cuyo alias es `auth`. Si el usuario que accede no está validado se le redirigirá a la ruta `auth/login`, si deseamos cambiar esta dirección o realizar otra acción podemos modificar este filtro.

Para utilizar este *middleware* tenemos que editar el fichero `app/Http/routes.php` y modificar las rutas o grupos de rutas que queramos proteger, por ejemplo:

```
// Para proteger una clausula del fichero de rutas:
Route::get('admin/catalog', ['middleware' => 'auth', function() {
    // Solo se permite el acceso a usuarios autenticados
}]);

// Para proteger una acción de un controlador:
Route::get('profile', [
    'middleware' => 'auth',
    'uses' => 'ProfileController@show'
]);

// O por ejemplo para proteger un grupo de rutas:
Route::group(['middleware' => 'auth'], function()
{
    Route::get('catalog',        'CatalogController@getIndex');
    Route::get('catalog/create', 'CatalogController@getCreate');
});
```

Si lo deseamos también podemos especificar el uso de este *middleware* desde el constructor del controlador:

```
public function __construct()
{
    $this->middleware('auth');
}
```

Sin embargo, lo más recomendable será indicarlo desde el fichero de rutas pues así tendremos todas las rutas y filtros centralizados en un único fichero.

Ejercicios

En los ejercicios de esta sección vamos a completar el proyecto del videoclub terminando el procesamiento de los formularios y añadiendo el sistema de autenticación de usuarios.

Ejercicio 1 - Migración de la tabla usuarios (0.5 puntos)

En primer lugar vamos a crear la tabla de la base de datos para almacenar los usuarios que tendrán acceso a la plataforma de gestión del videoclub.

Como hemos visto en la teoría, Laravel ya incluye una migración con el nombre `create_users_table` para la tabla `users` con todos los campos necesarios. Vamos a abrir esta migración y a comprobar que los campos incluidos coinciden con los de la siguiente tabla:

Campo	Tipo	Modificador
id	Autoincremental	
name	String	
email	String	*unique*
password	String de longitud 60	
remember_token	Campo remember_token	
timestamps	Timestamps de *Eloquent*	

Comprueba también que en el método `down` de la migración se deshagan los cambios que se hacen en el método `up`, en este caso sería eliminar la tabla.

Por último usamos el comando de Artisan que añade las nuevas migraciones y comprobamos con PHPMyAdmin que la tabla se ha creado correctamente con todos campos indicados.

Ejercicio 2 - *Seeder* de usuarios (0.5 puntos)

Ahora vamos a proceder a rellenar la tabla `users` con los datos iniciales. Para esto editamos el fichero de semillas situado en `database/seeds/DatabaseSeeder.php` y seguiremos los siguientes pasos:

- Creamos un método privado (dentro de la misma clase) llamado `seedUsers()` que se tendrá que llamar desde el método `run` de la forma:

```
public function run()
{
  // ... Llamada al seed del catálogo

  self::seedUsers();
  $this->command->info('Tabla usuarios inicializada con datos!');
}
```

- Dentro del nuevo método `seedUsers()` realizamos las siguientes acciones:
  - En primer lugar borramos el contenido de la tabla `users`.
  - Y a continuación creamos un par de usuarios de prueba. Recuerda que para guardar el *password* es necesario encriptarlo manualmente usando el método `bcrypt` (Revisa la sección "Registro de un usuario").

Por último tendremos que ejecutar el comando de Artisan que procesa las semillas. Una vez realizado esto comprobamos en PHPMyAdmin que se han añadido los usuarios a la tabla *users*.

Ejercicio 3 - Sistema de autenticación (1 punto)

En este ejercicio vamos a completar el sistema de autenticación. En primer lugar editamos el fichero `app/Http/routes.php` y realizamos las siguientes acciones:

- Añade las rutas necesarias para realizar el *login* y *logout* (revisa la teoría). Dos de estas rutas ya las teníamos definidas para que devolvieran una vista, cambialas para que apunten a los métodos apropiados del controlador `AuthController` y añade también la ruta tipo POST que falta.
- Añadimos un *middleware* de tipo grupo que aplique el filtro `auth` para proteger todas las rutas menos la raíz `/` y la de `/auth/login`.

Modifica el controlador `AuthController` para que cuando se realice el *login* correctamente te redirija a la ruta `/catalog`. Para esto tienes que definir la propiedad `redirectPath` para añadir la ruta de redirección (revisa el apartado "Autenticación de un usuario" de la teoría).

Añade la vista asociada a la acción de *login*. Utiliza el fichero `login.blade.php` de las plantillas de los ejercicios. Solo tienes que copiarlo en la carpeta correcta y completar los TODOs.

A continuación abrimos el controlador `HomeController` para completar el método `getHome`. Este método de momento solo realiza una redirección a `/catalog`. Modifica el código para que en caso de que el usuario no esté autenticado le redirija a la ruta `/auth/login`

Comprueba en este punto que el sistema de autenticación funciona correctamente: no te permite entrar a la rutas protegidas si no estás autenticado, puedes acceder con los usuarios definidos en el fichero de semillas y funciona el botón de cerrar sesión.

Por último edita la vista `resources/views/partials/navbar.blade.php` que habíamos copiado de las plantillas y cambia la línea `@if( true || Auth::check() )` por `@if( Auth::check() )`. De esta forma el menú solo se mostrará cuando el usuario esté autenticado.

Ejercicio 4 - Añadir y editar películas (1 punto)

En primer lugar vamos a añadir las rutas que nos van a hacer falta para recoger los datos al enviar los formularios. Para esto editamos el fichero de rutas y añadimos dos rutas (también protegidas por el filtro `auth`):

- Una ruta de tipo POST para la url `catalog/create` que apuntará al método `postCreate` del controlador `CatalogController`.
- Y otra ruta tipo PUT para la url `catalog/edit/{id}` que apuntará al método `putEdit` del controlador `CatalogController`.

A continuación vamos a editar la vista `catalog/edit.blade.php` con los siguientes cambios:

- Revisar que el método de envío del formulario sea tipo PUT.
- Tenemos que modificar todos los *inputs* para que como valor del campo ponga el valor correspondiente de la película. Por ejemplo en el primer *input* tendríamos que añadir value="{{$pelicula->title}}". Realiza lo mismo para el resto de campos: *year*, *director*, *poster* y *synopsis*. El único campo distinto será el de *synopsis* ya que el *input* es tipo *textarea*, en este caso el valor lo tendremos que poner directamente entre la etiqueta de apertura y la de cierre.

Por último tenemos que actualizar el controlador `CatalogController` con los dos nuevos métodos. En ambos casos tenemos que usar la inyección de dependencias para añadir la clase `Request` como parámetro de entrada (revisa la sección "Datos de entrada" de la teoría). Además para cada método haremos:

- En el método `postCreate` creamos una nueva instancia del modelo `Movie`, asignamos el valor de todos los campos de entrada (*title, year, director, poster* y *synopsis*) y los guardamos. Por último, después de guardar, hacemos una redirección a la ruta `/catalog`.
- En el método `putEdit` buscamos la película con el identificador pasado por parámetro,

actualizamos sus campos y los guardamos. Por último realizamos una redirección a la pantalla con la vista detalle de la película editada.

> Nota: de momento en caso de error no se mostrará nada.

Capítulo 5.
Paquetes, Rest y Curl

En este capítulo en primer lugar vamos a ver como instalar paquetes adicionales en Laravel. Esta opción nos permitirá, de forma muy sencilla, ampliar la funcionalidad de nuestra web y aprovechar código de librerías de terceros.

A continuación se verá como crear una interfaz tipo RESTful para por ejemplo implementar una API y como probar estos métodos mediante cURL. Además también veremos como utilizar la autenticación HTTP básica para proteger una API (o una Web) con usuario y contraseña, y algunos tipos de respuestas especiales de Laravel que nos serán útiles a la hora de implementar una API.

Instalación de paquetes

Además de todas las utilidades que incorpora Laravel y que podemos utilizar directamente sin instalar nada más, también nos permite añadir de forma muy sencilla paquetes para complementar su funcionalidad.

Para instalar un paquete tenemos dos opciones, utilizar el comando de `composer`:

```
sudo composer require <nombre-del-paquete-a-instalar>
```

O editar el fichero `composer.json` de la raíz de nuestro proyecto, añadir el paquete en su sección "`require`", y por último ejecutar el comando:

```
sudo composer update
```

Esta última opción es más interesante ya que nos permitirá una mayor configuración, por ejemplo, indicar la versión del paquete o repositorio, etc.

La lista de todos los paquetes disponibles la podemos encontrar en "https://packagist.org", en la cual permite realizar búsquedas, ver el número de instalaciones de un paquete, etc.

Algunos de los paquetes más utilizados en Laravel son:

Nombre	Descripción	Url
Extended Generators	Extensión de la Auto-generación de código	https://github.com/laracasts/Laravel-5-Generators-Extended
Debugbar	Barra de depuración de Laravel	https://github.com/barryvdh/laravel-debugbar
IDE Helper	Helper para IDEs	https://github.com/barryvdh/laravel-ide-helper
Entrust	Role permissions for Laravel 5	https://github.com/Zizaco/entrust
Ardent	Modelos auto-validados	https://github.com/laravelbook/ardent
MongoDB	Extensión para soportar MongoDB	https://github.com/jenssegers/laravel-mongodb
Notification	Notificaciones	https://github.com/edvinaskrucas/notification
Former	Automatización de formularios	https://github.com/anahkiasen/former
Image	Manipulación de imágenes	https://github.com/Intervention/image
Stapler	Manipulación de ficheros	https://github.com/CodeSleeve/laravel-stapler
User Agent	Obtener info. del *user agent*	https://github.com/jenssegers/laravel-agent
Sitemap	Genearción del Sitemap	https://github.com/RoumenDamianoff/laravel-sitemap
Excel	Trabajar con Excel y csv	https://github.com/Maatwebsite/Laravel-Excel
DOMPdf	Trabajar con PDF	https://github.com/barryvdh/laravel-dompdf

Una vez añadido el paquete tendremos que modificar también el fichero de configuración `config/app.php`, en su sección `providers` y `aliases`, para que la plataforma encuentre el paquete que acabamos de añadir. Los detalles de configuración en general se encuentran indicados en la web o repositorio de GitHub de cada proyecto.

Ejemplo: instalación de paquete de notificaciones

Por ejemplo, para instalar el paquete "Notification" de *edvinaskrucas* tendríamos que editar el fichero `composer.json` y en su sección `require` añadir la siguiente línea:

```
"edvinaskrucas/notification": "5.*"
```

> Al añadir el paquete al fichero `composer.json` tenemos que llevar mucho cuidado de añadir una coma como separador de los elementos de la sección `require`, **exceptuando** el último elemento de la lista, que no tendrá que llevar coma al final (sino nos dará error).

Una vez añadido ejecutamos el siguiente comando para instalar el paquete:

```
sudo composer update
```

Y después de instalar tendríamos que editar el fichero `config/app.php` para añadir, en la sección `providers`, la siguiente línea:

```
Krucas\Notification\NotificationServiceProvider::class,
```

Y en la sección `aliases` lo siguiente:

```
'Notification' => Krucas\Notification\Facades\Notification::class,
```

Además también tenemos que añadir el *middleware* al fichero `app/Http/Kernel.php`. En el array de `$middleware` añadimos la siguiente línea después de `'Illuminate\Session\Middleware\StartSession'`:

```
\Krucas\Notification\Middleware\NotificationMiddleware::class,
```

Con esto ya tendríamos instalada esta librería y podríamos empezar a utilizarla. Por ejemplo, para añadir una notificación desde un controlador podemos utilizar los siguientes métodos:

```
Notification::success('Success message');
Notification::error('Error message');
Notification::info('Info message');
Notification::warning('Warning message');
```

> Importante: tenemos que acordarnos de añadir el espacio de nombres de esta clase (`use Notification;`) en el controlador donde la vayamos a utilizar, de lo contrario nos aparecería un error.

Y para mostrar las notificaciones, desde una vista (preferiblemente desde el *layout* principal), añadiríamos el siguiente código:

```
{!! Notification::showAll() !!}
```

O también podemos usar la extensión que incorpora para Blade y simplemente añadir:

```
@notification()
```

Controladores de recursos *RESTful*

Laravel incorpora un tipo especial de controlador, llamado controlador de recuso (*recource controller*), que facilita la construcción de controladores tipo *RESTful*. Para esto simplemente tendríamos que usar el comando de Artisan `php artisan make:controller <nombre-controlador>` para crear el controlador y añadir la ruta al fichero de rutas `routes.php` usando `Route::resource`.

Por ejemplo, para crear un controlador para la gestión de imágenes almacenadas en la aplicación, en primer lugar ejecutaríamos el siguiente comando:

```
php artisan make:controller PhotoController
```

Esto crearía el controlador `PhotoController` (incluyendo todos los métodos necesarios) en la carpeta `app/Http/Controllers`. Lo único que nos faltaría es registrar las rutas asociadas añadiendo al fichero `app/Http/routes.php` la siguiente línea:

```
Route::resource('photo', 'PhotoController');
```

Esta línea de ruta crea por si sola múltiples rutas para gestionar todos los tipos de peticiones *RESTful*. Además, el controlador creado mediante Artisan estará preparado con todos los métodos necesarios para responder a todos los tipos de peticiones. En la siguiente tabla se muestra un resumen de todas las rutas generadas, el tipo de petición a la que responden y la acción que realizan en el controlador:

Verbo	Ruta	Acción	Controlador / método
GET	/photo	index	PhotoController@index
GET	/photo/create	create	PhotoController@create
POST	/photo	store	PhotoController@store
GET	/photo/{resource}	show	PhotoController@show
GET	/photo/{resource}/edit	edit	PhotoController@edit
PUT/PATCH	/photo/{resource}	update	PhotoController@update
DELETE	/photo/{resource}	destroy	PhotoController@destroy

Restringir rutas en un controlador RESTful

En ocasiones nos interesará declarar solamente un subconjunto de las acciones que soporta REST, para esto, al declarar la ruta tipo `resource` tenemos que añadir un tercer parámetro con la opción `only` (para que solo se creen esas rutas) o `except` (para que se creen todas las rutas excepto las indicadas), por ejemplo:

```
Route::resource('photo', 'PhotoController',
                ['only' => ['index', 'show']]);

Route::resource('photo', 'PhotoController',
                ['except' => ['create', 'store', 'update', 'destroy']]);
```

Los métodos que no utilicemos los podremos borrar del código generado para el controlador.

Middleware

Para añadir *middleware* a un controlador tipo recurso tenemos dos opciones. La primera es definir un grupo que englobe a la ruta, por ejemplo:

```
Route::group(['middleware' => 'auth'], function()
{
    Route::resource('photo', 'PhotoController',
                    ['except' => ['index', 'show']]);
});
```

La otra opción es definir el *middleware* en el constructor de la controlador de la forma:

```
class AwesomeController extends Controller
{
    public function __construct()
    {
        $this->middleware('auth');
        $this->middleware('log', ['only' => ['store', 'update', 'destroy']]);
    }
}
```

Controladores de recursos anidados

Para "anidar" controladores de recursos se tiene que utilizar el punto "." como separador de los recursos en la declaración de la ruta, por ejemplo:

```
Route::resource('photos.comments', 'PhotoCommentController');
```

Esta ruta representaría que el recurso "*comments*" estaría anidado o contenido en el recurso "*photos*".

Las rutas generadas para este tipo de recursos siguen el siguiente patrón: `photos/{photoResource}/comments/{commentResource}` , donde se tiene que especificar el identificador de ambos recursos para poder acceder.

Los controladores asociados recibirían en este caso dos identificadores, primero el del recurso base y segundo el del recurso anidado:

```
class PhotoCommentController extends BaseController
{
    public function show($photoId, $commentId)
    {
        //...
    }
}
```

Rutas adicionales en un controlador tipo RESTful

Si queremos definir rutas adicionales para un controlador de recursos simplemente las tenemos que añadir al fichero de rutas `routes.php` antes que las rutas del propio recurso, por ejemplo:

```
Route::get('photos/popular', 'PhotoController@getPopular');
Route::resource('photos', 'PhotoController');
```

Probar nuestra API con *cURL*

Para probar una API lo podemos hacer fácilmente utilizando el comando `curl` desde consola, el cual permite enviar peticiones de cualquier tipo a una URL, especificar las cabeceras, parámetros, etc.

Por ejemplo, para realizar una petición tipo GET a una URL simplemente tenemos que hacer:

```
$ curl -i http://localhost/recurso

HTTP/1.1 200 OK
Transfer-Encoding: chunked
Date: Fri, 27 Jul 2012 05:11:00 GMT
Content-Type: text/plain

¡Hola Mundo!
```

Donde la opción `-i` indica que se muestren las cabeceras de la respuesta.

Opcionalmente, al hacer la petición podemos indicar las cabeceras con el parámetro `-H`. Por ejemplo, para solicitar datos en formato JSON tenemos que hacer:

```
$ curl -i -H "Accept: application/json" http://localhost/recurso

HTTP/1.1 200 OK
Date: Fri, 27 Jul 2012 05:12:32 GMT
Cache-Control: max-age=42
Content-Type: application/json
Content-Length: 27

{
    "text": "¡Hola Mundo!"
}
```

Como hemos visto por defecto se realiza una petición tipo GET. Si queremos realizar otro tipo de petición lo tendremos que indicar con el parámetro `-X` seguido del método a utilizar (POST, PUT, DELETE). Además, con la opción `-d` podemos añadir los parámetros de la petición. Los parámetros tendrán que ir entre comillas y en caso de indicar varios los separaremos con `&`. Por ejemplo, para realizar una petición tipo POST con dos parámetros:

```
$ curl -i -H "Accept: application/json" -X POST
        -d "name=javi&phone=800999800" http://localhost/users
```

De la misma forma podemos hacer una petición tipo PUT (para actualizar datos) o tipo DELETE (para eliminarlos). Por ejemplo:

```
$ curl -i -H "Accept: application/json" -X PUT
        -d "name=new-name" http://localhost/users/1

$ curl -i -H "Accept: application/json" -X DELETE http://localhost/users/1
```

Para añadir más de una cabecera tenemos que indicar varias veces la opción `-H`, por ejemplo:

```
$ curl -i -H "Accept: application/json"
        -H "Content-Type: application/json" http://localhost/resource

$ curl -i -H "Accept: application/xml"
        -H "Content-Type: application/xml" http://localhost/resource
```

Por ejemplo, si queremos realizar una petición tipo POST que envíe código JSON y que también espere la respuesta en JSON tendríamos que indicar ambas cabeceras y añadir el JSON que queramos en los parámetros con `-d` de forma normal:

```
$ curl -i -H "Accept: application/json"
        -H "Content-Type: application/json" -X POST
        -d '{"title":"xyz","year":"xyz"}' http://localhost/resource
```

Como resumen, las opciones más importantes de `curl` son:

Opción	Descripción
`-i`	Mostrar las cabeceras de respuesta
`-H "header"`	Configurar las cabeceras de la petición
`-X <type>`	Indicar el método de la petición: POST, PUT, DELETE. Si no indicamos nada la petición será de tipo GET.
`-d "params"`	Añadir parámetros a la petición. Los parámetros tendrán que ir entre comillas "". Si queremos pasar varios parámetros utilizaremos como separador " `&` "

Plugins o extensiones

Además de `curl` también podemos utilizar otro tipo de programas para probar una API. En Firefox y Chrome podemos encontrar extensiones que nos facilitarán este tipo de trabajo. Por ejemplo, en Firefox podemos encontrar:

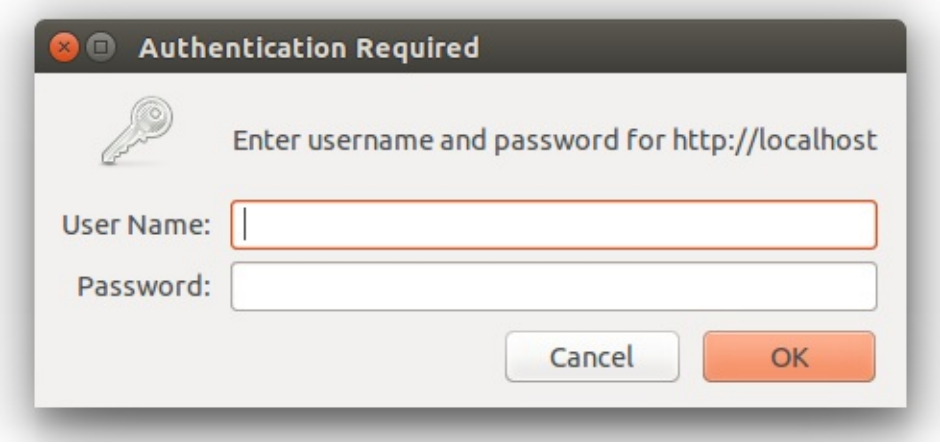
- *HttpRequester*: https://addons.mozilla.org/en-US/firefox/addon/httprequester/
- *Poster*: https://addons.mozilla.org/en-US/firefox/addon/poster/

Una vez instalado los podremos abrir desde el menú `Herramientas`, y utilizarlo a través de una interfaz visual muy sencilla.

Para Chrome también podemos encontrar muchas extensiones si buscamos en su tienda (https://chrome.google.com/webstore/category/apps). Algunas opciones interesantes son "*Advanced REST client*" o "*Postman - REST Client*".

Autenticación HTTP básica

El sistema de autenticación HTTP básica proporciona una forma rápida de identificar a los usuarios sin necesidad de crear una página con un formulario de login. Este sistema, cuando se accede a través de la web, automáticamente mostrará una ventana emergente para solicitar los datos de acceso:



Pero es más común su utilización para proteger las rutas de una API. En este caso las credenciales se tendrían que enviar en la cabecera de la petición.

Para proteger una ruta usando el sistema de autenticación básico simplemente tenemos que añadir el filtro llamado `auth.basic` a la ruta o grupo de rutas, de la forma:

```
Route::get('profile', ['middleware' => 'auth.basic', function()
{
    // Zona de acceso restringido
}]);
```

Por defecto este filtro utiliza la columna `email` de la tabla de usuarios para la validación.

Una vez superada la autenticación básica se crea la sesión del usuario y en cliente se almacenaría una *cookie* con el identificador de la sesión.

Autenticación HTTP básica sin estado

Si no queremos que la sesión se mantenga y que no se almacene una *cookie* tenemos que utilizar el sistema de autenticación "sin estado". Este sistema lo que realiza es simplemente solicitar siempre el usuario y contraseña y no almacenar las credenciales del usuario. Esta opción se suele utilizar mucho para la implementación de una API.

Laravel no trae un *Middleware* por defecto para la autenticación sin estado pero lo podemos crear rápidamente. En primer lugar ejecutamos el comando de Artisan para crear un nuevo *middleware*:

```
php artisan make:middleware AuthenticateOnceWithBasicAuth
```

A continuación editamos la nueva clase creada para que tenga el siguiente contenido:

```
<?php
namespace Illuminate\Auth\Middleware;

use Auth;
use Closure;

class AuthenticateOnceWithBasicAuth
{
    public function handle($request, Closure $next)
    {
        return Auth::onceBasic() ?: $next($request);
    }
}
```

Básicamente lo que tenemos que añadir es el espacio de nombres `use Auth;` y la línea que realiza la validación dentro de la función `handle`. La función `onceBasic` lanza la autenticación y en caso de que sea correcta permitirá continuar con la petición, y en otro caso devolverá un error de autenticación.

Por último nos faltaría registrar el *middleware* para poder utilizarlo. Para esto abrimos el fichero `app/Http/Kernel.php` y añadimos la siguiente línea al array de `routeMiddleware`:

```
'auth.basic.once' => \App\Http\Middleware\AuthenticateOnceWithBasicAuth::class,
```

Como se puede ver le hemos asignado el alias `auth.basic.once`, así que ya podemos usarlo para añadir la autenticación HTTP básica sin estado a nuestras rutas:

```
Route::get('api/user', ['middleware' => 'auth.basic.once', function() {
    // ...
}]);
```

Pruebas con *cURL*

Si intentamos acceder a una ruta protegida mediante autenticación básica utilizando los comando de *cURL* que hemos visto obtendremos el siguiente error:

```
HTTP/1.1 401 Unauthorized
```

Pero *cURL* permite también indicar el usuario y contraseña añadiendo el parámetro `-u` o también `--user` (equivalente):

```
$ curl --user username:password http://localhost/recurso
$ curl -u username:password http://localhost/recurso
```

> Si solamente indicamos el usuario (y no el password) se nos solicitará al pulsar ENTER, y además al introducirlo no se verá escrito en la pantalla.

Convertir modelos en Arrays o JSON

Laravel incluye métodos para transformar fácilmente el resultado obtenido de la consulta a un modelo de datos a formato JSON o a formato array. Esto es especialmente útil cuando estamos diseñando una API y queremos enviar los datos en formato JSON. Además, al realizar la transformación se incluirán los datos de las relaciones que se hayan cargado al hacer la consulta. Para realizar esta transformación simplemente tenemos que usar los métodos `toJson()` o `toArray()` sobre el resultado de la consulta:

```
$user = User::first();

$arrayUsuario = $user->toArray();

$jsonUsuario = $user->toJson();
```

También podemos transformar una colección entera de datos a este formato, por ejemplo:

```
$arrayUsuarios = User::all()->toArray();

$jsonUsuarios = User::all()->toJson();
```

En ocasiones nos interesará ocultar determinados atributos de nuestro modelo en la conversión a array o a JSON, como por ejemplo, el password o el identificador. Para hacer esto tenemos que definir el campo protegido `hidden` de nuestro modelo con el array de atributos a ocultar:

```
class User extends Model {

    protected $hidden = ['id', 'password'];

    // O también podemos indicar solamente aquellos que queramos mostrar:
    // protected $visible = ['name', 'address'];

}
```

Respuestas especiales

Con lo que hemos visto en la sección anterior solo se realizaría la transformación a JSON, pero si queremos devolverlo como respuesta de una petición (por ejemplo, como respuesta a una petición de un método de una API), tendremos que utilizar el método `response()-`

`>json()` , el cual además añadirá en la cabecera de la respuesta que los datos enviados están en formato JSON. Por ejemplo:

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

return response()->json( $usuarios );
```

Este método también puede recibir otro tipo de valores (como una variable, un array, etc.) y los transformará también a JSON para devolverlos como respuesta a la petición:

```
return response()->json( ['name' => 'Steve', 'state' => 'CA'] );
```

Si queremos especificar el código de la respuesta, por ejemplo cuando queremos indicar que ha sucedido algún error, podemos añadirlo como segundo parámetro, por ejemplo:

```
return response()->json( ['error'=>true, 'msg'=>'Error al procesar la petición' ], 500 );
```

La lista completa de los códigos que podemos utilizar la podéis encontrar en:

http://es.wikipedia.org/wiki/Anexo:C%C3%B3digos_de_estado_HTTP

Ejercicios

En esta sección de ejercicios vamos a terminar la web de gestión del videoclub añadiendo notificaciones, el funcionamiento de algunos botones que faltaban y por último, una API tipo RESTful para el acceso externo.

Ejercicio 1 - Notificaciones (0.5 puntos)

En este primer ejercicio vamos a instalar una librería externa para mostrar las notificaciones de nuestra aplicación, para esto tenéis que seguir los pasos indicados en el apartado de teoría "Ejemplo: instalación de paquete de notificaciones".

Una vez instalado y correctamente configurado vamos a modificar los controladores para mostrar un aviso tipo `success` después de guardar y editar una película. La notificación se tendrá que añadir antes de realizar la redirección. Le podéis poner los textos: "La película se ha guardado/modificado correctamente".

Por último vamos a modificar la vista con el *layout* principal, situada en `resources/views/layouts/master.blade.php`, para indicar que se muestren las notificaciones justo antes del contenido principal:

```
<div class="container">
    @notification()

    @yield('content')
</div>
```

Ejercicio 2 - Completando botones (1 punto)

En este ejercicio vamos añadir la funcionalidad de los botones de alquilar, devolver y eliminar película. Todos estos botones están situados en la vista detalle de una película (el de eliminar lo tendremos que añadir). En todos los casos tendremos que crear una nueva ruta, un nuevo método en el controlador, actualizar el botón en la vista y mostrar una notificación después de realizar la acción. En la siguiente tabla se muestra un resumen de las rutas:

Ruta	Tipo	Controlador / Acción
/catalog/rent/{id}	PUT	CatalogController@putRent
/catalog/return/{id}	PUT	CatalogController@putReturn
/catalog/delete/{id}	DELETE	CatalogController@deleteMovie

En primer lugar tenéis que añadir las rutas al fichero `routes.php` y posteriormente modificar el controlador `CatalogController` para añadir los tres nuevos métodos. Estos tres métodos son similares al método que ya habíamos implementado antes para editar los datos de una película. En el caso de `putRent` y `putReturn` únicamente modificaremos el campo `rented` asignándole el valor *true* y *false* respectivamente, y una vez guardado añadiremos la notificación y realizaremos una redirección a la pantalla con la vista detalle de la película. En el método `deleteMovie` también obtendremos el registro de la película pero tendremos que llamar al método `delete()` de la misma, una vez hecho esto añadiremos la notificación y realizaremos una redirección al listado general de películas.

A continuación tenemos que editar la vista detalle de películas para modificar los botones (`resources/views/catalog/show.blade.php`). Dado que la acciones se tienen que realizar usando peticiones HTTP tipo PUT y DELETE no podemos poner un enlace normal (ya que este sería tipo GET). Para solucionarlo tenemos que crear un formulario alrededor del botón y asignar al formulario el método correspondiente, por ejemplo:

```
<form action="{{action('CatalogController@putReturn', $pelicula->id)}}"
    method="POST" style="display:inline">
    {{ method_field('PUT') }}
    {!! csrf_field() !!}
    <button type="submit" class="btn btn-danger" style="display:inline">
        Devolver película
    </button>
</form>
```

Ejercicio 3 - Api Restful y pruebas (1.5 puntos)

En este ejercicio vamos a crear una API tipo *RESTful* para permitir el acceso y gestión del catálogo del videoclub de forma externa. En la siguiente tabla se muestra el listado de todas las rutas que vamos a definir para la API:

Ruta	Método	Filtro	Controlador / Método
/api/v1/catalog	GET		APICatalogController@index
/api/v1/catalog/{id}	GET		APICatalogController@show
/api/v1/catalog	POST	auth.basic.once	APICatalogController@store
/api/v1/catalog/{id}	PUT	auth.basic.once	APICatalogController@updat
/api/v1/catalog/{id}	DELETE	auth.basic.once	APICatalogController@destro
/api/v1/catalog/{id}/rent	PUT	auth.basic.once	APICatalogController@putRe
/api/v1/catalog/{id}/return	PUT	auth.basic.once	APICatalogController@putRe

Como se ve en la tabla, tendremos que definir todas las rutas *RESTful* para el catálogo, además de dos especiales: `/rent` y `/return`. Todas las rutas estarán protegidos con contraseña (usando autenticación HTTP básica **sin estado**) a excepción de `index` y `show` que serán públicas. Tenéis que comprobar que las rutas y filtros sean los correctos usando el método de Artisan `php artisan route:list`.

> Pista 1: El *middleware* "auth.basic.once" de autenticación básica sin estado no viene definido con Laravel por defecto. Para añadirlo tenéis que seguir las instrucciones indicadas en el apartado "Autenticación HTTP básica sin estado" de la teoría.
>
> Pista 2: Para poder aplicar un filtro solamente a algunos de los métodos del controlador tendréis que separar la declaración de las rutas. Para esto podéis utilizar el tercer parámetro con las opciones `only` y `except`.

A continuación tenéis que añadir el nuevo controlador `APICatalogController` usando el comando de Artisan que genera el controlador y todos los métodos RESTful. Las acciones y contenidos de los métodos serán muy similares a los de `CatalogController`, pero teniendo en cuenta que **no** tendremos que devolver una vista sino directamente el contenido de la consulta en formato JSON. Por ejemplo, el método que devuelve el listado de todas las películas sería simplemente:

```
public function index()
{
    return response()->json( Movie::all() );
}
```

Para devolver una respuesta en los métodos que realizan alguna acción (por ejemplo para indicar que la película se ha marcado como alquilada o que se ha modificado correctamente) podemos realizar lo siguiente:

```
public function putRent($id)
{
    $m = Movie::findOrFail( $id );
    $m->rented = true;
    $m->save();

    return response()->json( ['error' => false,
                            'msg' => 'La película se ha marcado como alquilada' ] );
}
```

Por último, utiliza `CURL` para comprobar que todas las rutas que has creado funcionan correctamente. Recuerda que para enviar

```
curl -i -H "Accept: application/json" -H "Content-Type: application/json"
        -X PUT -d '{"title":"nuevo titulo"}' http://localhost/catalog/21
```

> Aviso: hemos de tener cuidado con el método de actualizar los datos de una película, ya que los campos que no se envíen se asignarán como vacíos. Para solucionar esto podemos actualizar solamente los campos que contengan algún valor o enviar siempre todos los campos.