Frameworks: Laravel

(UD 5)

Contenido

Índice de contenido

Contenido

1. Características de Laravel

1. Características de Laravel

Ventajas de Laravel

1. Sintaxis simple y elegante.
2. Mapeo objeto-relacional (ORM): Eloquent.
3. Potente sistema de plantillas para vistas: Blade.
4. Reutiliza y moderniza componentes de Symfony.
5. Sencillo y potente.
6. Uso creciente en la industria: alternativa de futuro.
7. Comunidad de usuarios altamente especializada (buena relación señal/ruido… de momento)

Inconvenientes de Laravel

1. Instalación, configuración y despliegue complejos, incluso a través de VM.
2. Curva de aprendizaje elevada.
3. Se mueve según los intereses personales de su autor (es obra individual), con actualizaciones muy frecuentes y cambios caprichosos.
4. Inestabilidad de varios de sus componentes: a menudo hay que recurir a *fixes*.
5. Fuerte dependencia de la consola de comandos y de herramientas de terceros (composer, virtualbox, vagrant…) (Esto solo es un inconveniente para algunas personas)

Contenido

2. Instalación de Laravel

A) En un servidor local

B) Con Homestead (Vagrant)

C) Con Docker

2. Instalación de Laravel

A) Instalación en un servidor local (1/2)

1. Requiere una fuerte configuración del servidor: OpenSSL, Mbstring, Tokenizer. Y composer.
2. Necesita composer para resolver dependencias PHP e instalarse.
3. Descargar Laravel:

   $ composer global require "laravel/installer"

4. Instalar copia de Laravel limpia y lista para usar (hacerlo en un directorio accesible por Apache)

   (Puede requerir añadir el comando laravel al PATH del sistema)

   $ laravel new <nombre-aplicación>

   Por ejemplo: **$ laravel new blog**

2. Instalación de Laravel

A) Instalación en un servidor local (2/2)

- Más info para instalarlo en Linux:

  https://laravel.com/docs/5.X#installing-laravel

- Más info para instalarlo en Windows com XAMPP:

  http://developando.com/blog/laravel-composer-xampp-windows

- No olvides dar los permisos necesarios a tus archivos y directorios.

2. Instalación de Laravel

B) Instalación de Laravel con Homestead (1/4)

- **Vagrant** es una herramienta para crear un entorno de desarrollo virtual sin necesidad de instalar componentes en nuestra máquina.
- Cada entorno de desarrollo se llama **Box** y hay cientos de ellos.
- **Homestead** es un Box de Vagrant con todo lo necesario para desarrollar con Laravel sin necesidad de configurar nuestro servidor local.
- Este Box puede configurarse para equipararlo al servidor donde se vaya a desplegar la aplicación en el futuro.

2. Instalación de Laravel

B) Instalación de Laravel con Homestead (2/4)

1) Instalar Virtual Box 5.2 o superior
2) Instalar Vagrant
3) Ejecutar: **$ vagrant box add laravel/homestead**
4) Ejecutar: **$ git clone https://github.com/laravel/homestead.git Homestead**
5) Ejecutar: **$ cd Homestead**
6) Ejecutar: **$ bash init.sh** (o **init.bat** en Windows)
7) Editar Homestead.yaml para configurar nuestra máquina virtual
8) Editar archivo hosts (/etc/hosts o C:\Windows\System32\drivers\etc\hosts) para añadir: 192.168.10.10 homestead.test (o la IP que hayamos configurado en el archivo .yaml)
9) Ejecutar: **$ vagrant up** (desde el directorio Homestead) (la primera vez puede tardar un rato mientras crea y configura la VM)

B) Instalación de Laravel con Homestead (3/4)

10) Probar en el navegador: **http://homestead.test**

11) Configurar tu VM en el archivo Homestead.yaml

12) Puedes usar tu VM mediante ssh con el comando:

```
$ vagrant ssh
```

13) En tu máquina real, crea el directorio ~/code (o el que esté configurado en el archivo Homestead.yaml)

14) En la máquina virtual, ve al mismo directorio ~/code (o el que esté configurado en Homestead.yaml) y crea un proyecto Laravel con composer:

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

$ composer install
```

15) ¡Y listo! Trabajarás en tu directorio local ~/code. Cualquier cambio se mapeará al directorio ~/code en la VM. Mantén una consola abierta en la VM para ejecutar comandos artisan.

B) Instalación de Laravel con Homestead (4/4)

Nota: es posible que necesites generar una clave SSH para conexiones remotas seguras con tu servidor virtual.

- **Cómo hacerlo en Linux:**

  https://stackoverflow.com/questions/44463987/homestead-installation

- **Cómo hacerlo en Windows 10:**

  https://medium.com/@eaimanshoshi/i-am-going-to-write-down-step-by-step-procedure-to-setup-homestead-for-laravel-5-2-17491a423aa)

Más info en: https://laravel.com/docs/5.X/homestead

2. Instalación de Laravel

C) Laravel con Docker (1/2)

- **Docker** es una solución de virtualización aún más ligera que Vagrant.
- En Docker se virtualizan solo los componentes software que nuestra aplicación necesita para funcionar, y se ejecutan directamente en la máquina anfitrión.
- Los componentes se llaman **images**. Cuando se instancian, pasan de denominarse **containers**.
- Hay docker images para todo lo imaginable. Puedes encontrarlas en https://hub.docker.com
- Aunque Laravel recomienda la virtualización mediante Vagrant, cada vez más desarrolladores optan por Docker por su mayor simplicidad y eficiencia.

C) Laravel con Docker (2/2)

1) Instalar **docker** (y, probablemente, **docker-compose**) en tu servidor.
2) Buscar una imagen de Laravel en https://hub.docker.com. Por ejemplo, suelen funcionar muy bien las imágenes de Bitnami (una empresa propiedad de VMWare).
3) Crear un directorio para nuestra aplicación web. No importa en qué parte del árbol de directorios esté.
4) En la consola, teclear el comando para crear una copia local de las imágenes necesarias. En el caso de la imagen Laravel de Bitnami, el comando es:

   $ docker pull bitnami/laravel

5) Seguir las instrucciones para el despliegue del contenedor que el desarrollador ha debido colgar en hub.docker.com. En el caso de bitnami, hay que copiar un archivo remoto mediante el comando curl y levantar las imágenes docker con docker-compose.
6) **¡Y listo!** Nuestro Laravel estará funcionando con todas sus dependencias resueltas.

Contenido

3. Arquitectura de Laravel

3. Arquitectura de Laravel

Directorios tras una instalación limpia (algunos)

- **/document_root**
  - **composer.json:** info para composer.
  - **/app**: el código de nuestra aplicación. Modelos.
    - **/app/config**: configuración de la aplicación.
    - **/app/http**: peticiones HTTP, incluyendo los controladores.
  - **/database:** migraciones de la BD.
  - **/plugins**: pues eso.
  - **/resources:** assets, archivos de idioma y vistas.
  - **/storage:** caché, sesiones, vistas compiladas...
  - **/vendors**: librerías de terceros.

3. Arquitectura de Laravel

Convenciones en Laravel

- **Modelos**: Nombres igual que el de la tabla, en singular, en CamelCase y con mayúscula. Ejemplo: RegisteredUser
- **Controladores**: como el modelo, pero añadiendo la palabra "controller" o "controlador". Ejempo: RegisteredUserController
- **Métodos**: en camelCase empezando con minúscula. Ejemplo: User::getAll()
- **Atributos**: en snake_case empezando con minúscula. Ejemplo: User::first_name
- **Variables**: en camelCase empezando con minúscula. Ejemplos: bannedUsers (colección, en plural), articleContent (variable simple, en singular)
- **Tablas:** en plural y en snake_case. Ej: registered_users.
  - **Columnas**: en snake_case, sin referencia al nombre de la tabla. Ejemplo: first_name
  - **Clave primaria:** id (integer y auto-increment).
  - **Claves ajenas**: nombre tabla ajena en singlular + id. Ejemplo: article_id
  - **Timestamps**: created_at y updated_at
- **Tablas pivote:** en snake_case, en plural y orden alfabético. Ejemplo: articles_users será el pivote entre articles y users.

3. Arquitectura de Laravel

Variables de entorno: el archivo .env

- Este archivo contiene todas las variables de entorno que configuran la aplicación, como:
  - APP_ENV: ¿estamos en desarrollo o en producción?
  - APP_DEBUG: ¿mostrar errores para depuración?
  - APP_URL: URL base de la aplicación
  - DB_CONNECTION, DB_HOST, DB_USERNAME, etc.
- El archivo .env **NO debe sincronizarse con git** (o con el control de versiones que usemos) porque contiene información sensible.
- En los archivos de configuración haremos referencia a las variables de entorno. P. ej: en **/config/database.php** usaremos una expresión así:

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

- El primer parámetro de env() es la variable de entorno y el segundo es el valor por defecto en caso de que la variable no exista.

4. Artisan: la consola que mola

4. Artisan: la consola que mola

Qué es Artisan

- Artisan es una herramienta de consola que automatiza tareas habituales en Laravel como:
  - Generar esqueletos de controladores y modelos.
  - Crear migraciones de bases de datos (para manipular la estructura de las tablas)
  - Rellenar la BD con datos de prueba.
  - Hacer el enrutamiento.
  - Configurar la aplicación.
  - Crear baterías de pruebas.
  - Y otro montón de cosas.

4. Artisan: la consola que mola

Ejemplo de uso de Artisan: crear un controlador

```
$ php artisan make:controller Hola
```

- El controlador se crea en /app/Http/Controllers/Hola.php

4. Artisan: la consola que mola

Uso de artisan con virtualización

- ¡CUIDADO! Si utilizas Laravel con virtualización, tienes que hacer *login en la máquina virtual* para poder usar la consola.
- Con Vagrant:

  $ **vagrant ssh**

- Con Docker:

  $ docker exec -it <id-del-contenedor> bash

Contenido

5. Primeros pasos con Laravel: ¡Hola, mundo!

5: Primeros pasos. ¡Hola, mundo!

Paso 1: Crear una clave segura

```
$ php artisan key:generate
```

- Esto genera una clave aleatoria y la almacena en el archivo de configuración **/.env**
- Es necesaria para hacer conexiones remotas seguras con la aplicación.
- En versiones recientes de Laravel *es obligatorio crearla*.

5: Primeros pasos. ¡Hola, mundo!

Paso 2: Enrutamiento básico

/routes/web.php

(en versiones anteriores de Laravel: **/app/Http/routes.php**)

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

Probar en el navegador: **http://<mi-ruta>/index.php/hola**

5: Primeros pasos. ¡Hola, mundo!

Paso 3: Enrutar al controlador

/routes/web.php

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

ENRUTADOR

/app/Http/Controllers/Hola.php

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

CONTROLADOR

Probar en el navegador: **http://<mi-ruta>/index.php/hola**

5: Primeros pasos. ¡Hola, mundo!

Paso 4: Cargar una vista desde el controlador (1/2)

/routes/web.php

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

ENRUTADOR

/app/Http/Controllers/Hola.php

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

CONTROLADOR

/resources/views/hola.blade.php

```
<body>
    Hola {{$nombre}},
    ¿le apetece una partidita de ajedrez?
</body>
```

VISTA

Cargar una vista desde el controlador (2/2)

Probamos desde el navegador así:

http://<mi-ruta>/index.php/hola/ProfesorFalken

Contenido

6. Enrutamiento

Enrutamiento (1/5)

- Hay varias formas de generar una salida HTML desde en enrutador (ver el ejemplo *Hola mundo*):

/routes/web.php

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

// Forma 2: llamar a un controlador sin pasarle parámetros
Route::get('/hola', 'Hola@show');

// Forma 3: llamar a un controlador pasándole parámetros
Route::get('/hola/{nombre}', 'Hola@show');

// Forma 4: llamar a un controlador con un parámetro optativo
Route::get('/hola/{nombre?}', 'Hola@show');
```

Enrutamiento (2/5): rutas con nombre

- Se puede (y es **muy recomendable**) asignar un nombre a las rutas en el enrutador.
- Esto hace que más adelante podamos **cambiar la URL de los enlaces sin tener que modificar el código fuente** de nuestras vistas.

/routes/web.php

```
Route::get('/contactame', 'Controlador@contacto')->name('contact');
```

- Internamente esta ruta se direcciona con `route('contact')` pero el usuario verá la dirección http://servidor/contactame.
- En el futuro se puede cambiar la forma en la que lo ve el usuario, pero para el código fuente seguirá siendo `route('contact')`

Enrutamiento (3/5)

- Además de GET, en el enrutador se puede enrutar otras acciones:

```
Route::get();    // Solicitudes habituales

Route::post();   // Recepción de datos de formulario (para insert)

Route::put|patch();  // Recepción de datos para update

Route::delete(); // Recepción de datos para delete

Route::match(array('GET','POST'), 'ruta', acción)

                 // Responderá tanto a GET como a POST
```

- Los métodos de formulario PUT y DELETE no están soportados por la mayoría de los navegadores. Hay que emularlos así:

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

    @method('DELETE')
```

Enrutamiento (4/5)

- ¡CUIDADO! El orden de las rutas es importante. Por ejemplo, si pedimos la dirección http://<mi-servidor>/usuario/crear, esto fallará:

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

Route::get('usuario/crear', 'UsuarioController@create');
```

- El problema es que tratará de mostrar un contacto con nombre = "crear", porque la petición encaja con las dos rutas y **el enrutador siempre elegirá la primera ruta que encuentre**.

- La solución pasa por alterar el orden de las líneas en el enrutador:

```
Route::get('usuario/crear', 'UsuarioController@create');

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

Enrutamiento (5/5): Servidor RESTful

- Un **servidor RESTful** es aquel que responde a la arquitectura REST.
- Típicamente, el enrutador contendrá las 7 operaciones REST para cada recurso accesible desde la red.
- Por ejemplo, para un recurso llamado "usuario":

```
Route::get('usuario', 'UsuarioController@index')->name('user.index');

Route::get('usuario/{id}', 'UsuarioController@show')->name('user.show');

Route::get('usuario/crear', 'UsuarioController@create')->name('user.create');

Route::post('usuario/{id}', 'UsuarioController@store')->name('user.store');

Route::get('usuario/{id}/editar', 'UsuarioController@edit')->name('user.edit');

Route::patch('usuario/{id}', 'UsuarioController@update')->name('user.update');

Route::delete('usuario/{nick}', 'UsuarioController@destroy')->name('user.destroy');
```

Contenido

7. Vistas y plantillas: Blade

7. Plantillas y vistas

¿Qué es Blade?

- Blade es un poderoso y sencillo sub-lenguaje que nos permitirá generar plantillas para minimizar el código que necesitamos para nuestras vistas.
- Las plantillas Blade admiten condiciones y bucles para operar con las variables PHP, de modo que la misma plantilla se comporta de forma diferente con diferentes conjuntos de datos.
- ¡Y se acabó la pesadilla de abrir y cerrar comillas para concatenar variables a las salidas HTML!
- Además, unas plantillas pueden heredar de otras.

Blade: plantillas para vistas (1/8)

Ejemplo de **master layout**: /resources/views/layouts/master.blade.php

```
<html>
    <head>
        <title>@yield('Titulo')</title>
    </head>
    <body>
        @section('sidebar')
            Este es mi master sidebar.
        @show

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

"**yield**" significa "producir" o "generar"

La sección "**título**" está vacía en el master layout

La sección "**sidebar**" NO está vacía

La sección "**container**" vuelve a estar vacía

Blade: plantillas para vistas (2/8)

Ejemplo de **plantilla heredada**: /resources/views/page.blade.php

```
@extends('master')

@section('title', 'Titulo de la página')

@section('sidebar')
    <p>Esto se añadirá al master sidebar.</p>
@endsection

@section('content')
    <p>Aquí va el contenido de mi página.</p>
@endsection
```

Se puede crear el contenido de una sección con un string sencillo

Se puede añadir HTML a una sección que ya tenía contenido

Se puede crear el contenido de una sección con HTML

Blade: plantillas para vistas (3/8)

Cómo testear una vista rápidamente

En el enrutador /routes/web.php:

```
Route::get('blade', function () {
    return view('page');
});
```

En el navegador web:

http://<mi-ruta>/index.php/blade

Blade: plantillas para vistas (4/8)

Cómo pasar variables a las plantillas

Hay varias formas de enviar variables a una plantilla desde el enrutador o el controlador:

```
return view('vista', array('variable1'=>'valor', 'variable2'=>'valor'));
return view('vista', ['variable1'=>'valor', 'variable2'=>'valor']);
return view('vista')->with(['variable1'=>'valor', 'variable2'=>'valor']);
return view('vista', compact('variable1', 'variable2'));
```

Blade: plantillas para vistas (5/8)

Cómo usar las variables en las plantillas

Si tenemos esto en el enrutador /routes/web.php

```
Route::get('blade', function () {
    return view('page',array('name' => 'Manolo Escobar'));
});
```

...podemos usar la variable $name en la vista /resources/views/page.blade.php

```
@extends('layouts.master')
@section('title', 'Page Title')
@section('sidebar')
    <p>Esto se añadirá al sidebar del master layout.</p>
@endsection

@section('content')
  <h2>{{$name}}</h2>
  <p>Este es el contenido de mi página.</p>
@endsection
```

Blade: plantillas para vistas (6/8)

Cómo usar condicinales en las plantillas

En el enrutador /routes/web.php

```
Route::get('blade', function () {
    return view('page',array('name' => 'Manolo','day' => 'Viernes'));
});
```

En la vista /resources/views/page.blade.php

```
...todo el principio es igual...
@section('content')
    <h2>{{$name}}</h2>
    <p>Este es el contenido de mi página.</p>
    @if ($day == 'Viernes')
        <p>Hoy me roban el carro</p>
    @else
        <p>Hoy me dedico a cantar</p>
    @endif
@endsection
```

Blade: plantillas para vistas (7/8)

Cómo usar bucles en las plantillas

En el enrutador /routes/web.php

```
Route::get('blade', function () {
    $bebidas = array('Cerveza','Agua','Zumo');
    return view('page',array('name' => 'Manolo','day' => 'Lunes',
                             'bebidas' => $bebidas));
});
```

En la vista /resources/views/page.blade.php

```
...todo el principio es igual...
@section('content')
   ...aquí van los condicionales...
   <h2>Foreach Loop</h2>
   @foreach ($bebidas as $bebida)
      {{$bebida}} <br>
   @endforeach
@endsection
```

Blade: plantillas para vistas (8/8)

Cómo usar funciones PHP estándar en las plantillas

En la vista /resources/views/page.blade.php

```
<h2>Ejemplo de llamada a función estándar de PHP</h2>
<p>La fecha del servidor es: {{date(' D M, Y')}}</p>
```

Y RECUERDA: utilizar las llaves `{{...}}` en lugar de `<?php ... ?>`

- Hace que el código de la plantilla sea más limpio.
- Estemos protegidos contra ataques XSS.
- Y si alguna vez necesitamos saltarnos la protección contra XSS, podemos utilizar la sintaxis `{!!...!!}`

Blade: Principales directivas (1/2)

`@section` → Marca el inicio de una sección de contenido.

`@endsection` → Marca el final de una sección.

`@show` → Marca el final de una sección y la muestra.

`@yield('section')` → Muestra el contenido de una sección (si existe, claro).

`@extends('view')` → Hereda de una plantilla padre.

`@parent` → Muestra el contenido de la sección del mismo nombre en la plantilla padre.

`@include('view')` → Incluye una subvista.

`@if | @else | @endif` → Condicional.

`@for | @endfor` → Bucle for clásico.

`@foreach | @endforeach` → Buche foreach.

`@forelse | @empty | @endforelse` → Como foreach, pero con tratamiento de arrays que vienen vacíos.

Blade: Principales directivas (2/2)

`@break | @continue` → Para usar en los bucles.

`@switch | @case | @break` → Condicional múltiple.

`@isset($variable)` → Comprueba si la variable existe.

`@auth | @endauth` → La sección solo se muestra si hay un usuario autenticado en la aplicación.

`@php | @endphp` → Para añadir código PHP plano. Usar con moderación.

`{{- Comment -}}` → Comentarios (no serán renderizados).

`{{ $variable }}` → Equivalente a echo $variable, pero no es necesario abrir y cerrar PHP, y nos protegemos contra inyección de JS.

`{{ $variable ?? texto-por-defecto }}` → Como el anterior, pero chequea si la variable existe. Si existe, muestra su valor. Si no, muestra texto-por-defecto.

`$loop` → Es una variable muy útil para usar en y/o depurar un foreach. Nos dice si estamos en el primer elemento, en el último, cuántos loops llevamos, etc.

7. Plantillas y vistas

Blade: enviar formularios

Ejemplo: formulario de contacto que desamos almacenar en la BD:

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

En el enrutador /routes/web.php:

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

En el controlador:

```
public function store(Request $r) {
   $email = $r->get("email");
   $asunto = $r->get("asunto");
   ...etc...
```

Una forma alternativa:

```
public function store() {
   $email = request("email");
   $asunto = request("asunto");
   ...etc...
```

7. Plantillas y vistas

Blade: validar formularios

En el formulario:

```
<form method="POST" action="{{ route('mi-ruta') }}">
   @if ($errors->any())
      @foreach ($errors->all() as $error)
          {{ $error }}<br>
      @endforeach
   @endif
   <input type="email" name="email"><br>
   ...resto del formulario igua...
</form>
```

Todas las reglas de validación existentes están en:

https://laravel.com/docs/validation

En el controlador:

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

7. Plantillas y vistas

Blade: repopular formularios

En el formulario:

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

NOTA: el objeto **$errors** (disponible en todas las vistas) tiene muchos más métodos útiles. Puedes ver un resumen al final del tema.

Blade: enviar formularios PUT | PATCH | DELETE

- Los métodos PUT, PATCH y DELETE no están soportados por HTML (al menos, de momento).
- Pero el enrutador de Laravel sí los soporta.
- Para simularlos, debemos incluir un campo oculto en el formulario llamado *_method*

```
<form method="POST" action="{{ route('mi-ruta') }}">
   <input type="hidden" name="_method" value="PATCH">
   ...resto del formulario...
</form>
```

- O mejor todavía:

```
<form method="POST" action="{{ route('mi-ruta') }}">
   @method("PATCH")
</form>
```

Vistas: añadir CSS y JavaScript

- El **CSS** y el **JavaScript** que vayamos a usar deben colocarse en /public/css y /public/js
- Laravel ya trae dos archivos (app.css y app.js) basados en **Boostrap 4** para empezar a trabajar. Para usarlos, basta con añadir esto a la cabecera de nuestras vistas:

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

- Si queremos añadir cosas a nuestro CSS, NO debemos editar /public/app.css, porque es un CSS compilado y minimizado con **SASS**.
- Lo correcto para añadir nuestro CSS a ese archivo es:
  - Editar /resources/sass/app.css
  - Recompilar este archivo con SASS (o con **less** o con **stylus**)

`$ npm run dev` (o también: `$ yarn dev`)

- Y, por supuesto, siempre podemos crear nuestro propio CSS en otros archivos en /public/css

Vistas: personalizar errores

- Las vistas creadas en /resources/views/errors permiten personalizar fácilmente las pantallas de error HTTP.
- Por ejemplo, si creas un archivo /resources/views/errors/**404.blade.php**, esa vista se mostrará cada vez que ocurra un **error 404** (página no encontrada)

Contenido

8. Controladores

8. Controladores

Controladores en Laravel

- Los controladores en Laravel heredan de la clase Controller.
- Su nombre debería escribirse en singular, CamelCase y terminando en la palabra Controller. Por ejemplo:

  `UserController, LoginController, ArticleController`

- Cada función debe terminar en un **return**. Lo que la función devuelva será convertido automáticamente en una HTTP response 200.
- Si tras el return hay un array, se convertirá automáticamente en JSON.
- Se puede actuar sobre la HTTP response devolviendo esto

```
return response('contenido', 201)
        ->header('mi-cabecera', 'contenido')
        ->header('otra-cabecera', 'más-contenido')
        ->cookie('mi-cookie', 'valor');
        (etc)
```

8. Controladores

Controladores: tres formas de crearlos

Forma 1. Crear un controlador vacío. P. ej: un controlador para la tabla de usuarios:

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

Forma 2. Crear un controlador resource (con los métodos index(), create(), store(), show(), edit(), update() y destroy() del estándar REST)

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

En el enrutador (**routes/web.php**) se crearán *automáticamente* todas las rutas REST para este tipo de controlador si escribimos:

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

En nuestro ejemplo:

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

Controladores: tres formas de crearlos

Forma 3. Crear un controlador tipo API. Es como *resource*, pero sin create() ni edit(), porque una API no necesita mostrar los formularios de inserción/modificación.

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

En el enrutador (**routes/web.php**) se crean *automáticamente* todas las rutas para este tipo de controlador con:

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

Contenido

9. Migraciones

9. Migraciones

¿Qué son las migraciones?

- Las migraciones constituyen una especie de **control de versiones para la base de datos** de la aplicación.
- Permiten crear y modificar tablas de la BD con independencia del SGBD que estemos usando.
- Solo hay que editar estas variables de entorno en el archivo **.env** de Laravel.

```
DB_HOST=localhost

DB_DATABASE=mi-base-de-datos

DB_USERNAME=mi-usuario-de-BD

DB_PASSWORD=mi-password-de-BD
```

(Nota: Si te da un error de conexión desde artisan, aquí tienes la solución: https://kode-blog.io/laravel-5-migrations)

Cómo hacer migraciones (1/3)

Ejemplo: hacer la migración de una tabla llamada Users.

- **Paso 1**: crear la tabla de migraciones para Laravel (si ya está creada de antes, nos dará un error al intentar hacerlo otra vez):

```
$ php artisan migrate:install
```

- **Paso 2**: crear migración para la tabla users:

```
$ php artisan make:migration create_users_table
```

o bien:

```
$php artisan make:migration create_users_table --create
```

- El fichero se creará en /database/migrations y tendrá un *timestamp* en su nombre. Algo como:

```
/database/migrations/20201226072434createuserstable.php
```

- Si editas ese fichero, verás dos métodos:
  - up() → se ejecuta cuando se lanza la migración.
  - down() → se ejecuta cuando se cancela la migración.

Cómo hacer migraciones (2/3)

- **Paso 3**: editar el fichero **/database/migrations/<timestamp>createuserstable.php**:

```
public function up() {
    Schema::create('users', function (Blueprint $table) {
       $table->bigIncrements('id')->index();// UNSIGNED BIGINT AUTOINC.
       $table->string('name',75)->unique(); // VARCHAR
       $table->text('address')->nullable(); // TEXT
       $table->integer('level');            // INT
       $table->date('brith_date');          // DATE
       // La siguiente línea crea campos created_at y updated_at
       $table->timestamps();
    });
}
public function down() {
    Schema::drop('users');
}
```

Cómo hacer migraciones (3/3)

- **Paso 4**: lanzamos las migraciones:

  ```
  $ php artisan migrate
  ```

  Las tablas se crean en la BD y Laravel las tiene ahora fichadas para auto-crear los modelos.

- **Paso 5** (opcional): si necesitas revertir la creación de tablas:

  ```
  $ php artisan migrate:rollback
  ```

  O solo revertir el último paso en la creación de tablas

  - `$ php artisan migrate:rollback --step=1`
- O dejar la BD a su estado original (vacía) antes de reconstruirla:

  ```
  $ php artisan migrate:rollback
  ```

- **¡Cuidado! Estas acciones son destructivas**. Hay una forma de modificar una tabla sin borrarla y volver a crearla (v. pág. siguiente)

Migraciones: añadir columnas a una tabla

(por ejemplo, la columna *email* a la tabla *users*)

- **Paso 1**: ejecutar:

```
$ php artisan make:migration add_email_to_users --table=users
```

- **Paso 2**: editar /database/migration/add_email_to_users.php:

```
public function up() {
    Schema::table('users', function (Blueprint $table) {
        $table->string('email')->after('address');
    });
}
public function down() {
    Schema::table('users', function (Blueprint $table) {
        $table->dropColumn('email');
    });
}
```

Migraciones: otras operaciones

- Las migraciones pueden usarse para cualquier otra operación con la estructura de las tablas:
  - Cambiar tipos de columnas.
  - Cambiar atributos de columnas (null, unique, default…)
  - Cambiar o asignar claves primarias y ajenas.
- Esto permite reconstruir o actualizar la base de datos en cualquier momento sin necesidad de parchear la aplicación o exportar la BD a un archivo SQL para importarlo en otro servidor.
- Más info en: https://kode-blog.io/laravel-5-migrations

9. Migraciones

Seeding (1/2)

***Seeding* = Rellenar la BD con datos de prueba** (p.ej: la tabla *users*)

- **Paso 1**. Ejecutar:

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

- **Paso 2**. Editar /database/seeds/UsersTableSeeder.php:

```
public function run() {
    Users::truncate();   // Opativo: vacía la tabla antes de rellenarla
    DB::table('users')->insert([
        'name' => 'Stephen Falken',
        'address' => ' Oregon 97, Goose Island',
        'email' => 'sfalken@norad.com',
        'brith_date' => '1932-09-03',
    ]);
}
```

- **Paso 3**. Ejecutar:

```
$ php artisan db:seed --class=UsersTableSeeder
```

Seeding (2/2)

Cómo automatizar el seeding de varias tablas:

- **Paso 1**. Editar el fichero /database/seeds/DatabaseSeeder.php
- **Paso 2**. Añadir a la función **run()** una línea como esta por cada seeder que quiera ejecutar automáticamente:

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

- **Paso 3**. Ejecutar db:seed sin indicar la clase para lanzar todos los seeders en secuencia:

```
$ php artisan db:seed
```

9. Migraciones

Comandos útiles de artisan para migraciones

`$ php artisan migrate`
→ Lanza todas las migraciones.

`$ php artisan make:migration <nombre> --create=<tabla>`
→ Crea una migración para la tabla indicada.

`$ php artisan make:migration <nombre> --table=<tabla>`
→ Modifica una migración para la tabla indicada.

`$ php artisan migrate:rollback`
→ Retrocede UN paso en todas las migraciones.

`$ php artisan migrate:rollback --step=<N>`
→ Retrocede N pasos en todas las migraciones.

`$ php artisan migrate:reset`
→ Deshace todas las migraciones que se hayan ejecutado hasta ahora.

`$ php artisan migrate:refresh`
→ Reset + migrate en un solo comando.

`$ php artisan migrate:refresh --seed`
→ Reset + migrate + seed en un solo comando.

`$ php artisan migrate:fresh`
→ Elimina todas las tablas y lanza todas las migraciones.

`$ php artisan migrate:fresh --seed`
→ Elimina todas las tablas, lanza todas las migraciones y todos los seeders.

Contenido

10. Usando la BD con Eloquent

10. Usando la BD con Eloquent

Eloquent (1/6)

- **Eloquent es el componente de Laravel que permite manipular los datos de la BD.**
- Eloquent es un ORM (Object-Relational Mapping): una librería que mapea los objetos de nuestra aplicación con una BD relacional.
- Ejemplo de uso de Eloquent:

  Supongamos una tabla artículos(id, titulo, cuerpo). Con Eloquent, usar esa tabla desde un controlador es tan fácil como:

```
$art = Articulos::find('7');  // id del artículo buscado

echo $art->titulo;

$art->cuerpo = "Texto del cuerpo";

$art->save();
```

10. Usando la BD con Eloquent

Eloquent (2/6)

- **Paso 1**. Crear un modelo con el comando:

```
$ php artisan make:model <Mi-modelo>
```

  Por ejemplo:

```
$ php artisan make:model Articulo
```

  El modelo se crea en /app/Articulo.php

- **Nota**: si creas el modelo con la **opción -m**, se creará atomáticamente su migración, lo cual resulta muy práctico

```
$ php artisan make:model Articulo -m
```

- **Paso 2**. Editar el modelo /app/Articulos.php

```
protected $table = 'articulos';   // Nombre de la tabla
protected $primaryKey = 'id';     // Nombre de la clave primaria
protected $fillable = array('id','titulo','cuerpo'); // Campos de la tabla en
                                  // los que se permite la ASIGNACIÓN MASIVA
```

Eloquent (3/6)

- **Paso 3**. En el controlador, importamos el modelo:

```
use App\<nombre-del-modelo>;
```

Ahora el controlador ya puede acceder al modelo:

```
Articulos::all();           // Devuelve todos los artículos
Articulos::find($id);       // Devuelve el artículo con ese $id
```

- Eloquent proporciona un montón de métodos para consultar datos. Ejemplos:

```
Articulos::findOrFail($id); // Error 404 si el artículo no existe
Articulos::where('id', '>', 100)->get();  // Select con where
Articulos::where('id', '>', 100)->take(10)->get();
Articulos::max('id');
```

Eloquent (4/6)

- Para hacer un **INNER JOIN** podemos usar QueryBuilder (más detalles en el siguiente apartado):

```
$users = DB::table('users')
            ->join('contacts', 'users.id', '=', 'contacts.user_id')
            ->join('orders', 'users.id', '=', 'orders.user_id')
            ->select('users.*', 'contacts.phone', 'orders.price')
            ->get();
```

- También se puede hacer **LEFT JOIN** y **RIGHT JOIN**:

```
DB::table('A')->leftJoin('B'...);
DB::table('A')->rightJoin('B'...);
```

10. Usando la BD con Eloquent

Eloquent (5/6)

- También se puede **insertar** con Eloquent:

```
$art = new Articulo;
$art->titulo = 'Los Chitauri invaden Nueva York';
$art->cuerpo = 'Bla, bla, bla';
$art->save();
```

- O incluso hacer algo como esto (que insertará TODOS los campos del formulario que estén descritos en la propiedad **fillable** del modelo):

```
public function store(Request $request) {
   Articulo::create($request->all());
   return <vista>;
}
```

- Y, por supuesto, también podemos **modificar** y **borrar**:

```
$art = Articulos::find(18);     // MODIFICAR
$art->cuerpo = 'Nuevo cuerpo';
$art->save();
$art = Articulos::find(13);     // BORRAR
$art->delete();
```

10. Usando la BD con Eloquent

Eloquent (6/6) – Referencia esencial:

- **all()** → Recupera todos los registros de una tabla.
- **where("campo", valor)** → Aplica claúsula where.
- **orderBy("campo", "asc|desc")** → Aplica claúsula order by.
- **get()** → Recupera registros seleccionados. Se suele usar con where y/o order by:

```
Ciudades::where("ciudad", "Madrid")->orderBy("id", "asc")->get();
```

- **first()** → Recupera el primer registro.
- **latest()** → Recupera el último registro.
- **find(valor)** → Busca registros con ese valor en el campo id.
- **findOrFail(valor)** → Lanza un error 404 si no encuentra el registro.
- **count(), max(), min()…** → Utiliza funciones de agregado de SQL.
- **save()** → Inserta o actualiza registros.
- **update()** → Actualiza registros.
- **delete()** → Elimina registros.

10. Usando la BD con Eloquent

Relaciones entre tablas con Eloquent

- Las relaciones entre tablas también se definen con Eloquent.
- Aunque al principio parezca una forma rebuscada de hacerlo, luego ahorra mucho trabajo.
- Las relaciones, una vez definidas, se comportan como consultas y se puede operar con ellas como si lo fueran
- En los siguientes ejemplos, vamos a suponer que tenemos estas tablas:

```
usuarios(id#, nombre, passwd)
emails(id#, email, usuario_id) → Relación 1:1 con usuarios
articulos(id#, titulo, texto, idUsuario) → Relación 1:N con usuarios
roles(id#, nombre) → Relación N:N con usuarios
```

- ATENCIÓN: en la tabla “artículos” hemos usado a propósito un nombre no estándar para la clave ajena. La convención de Laravel es usuario_id, como en la tabla “emails”.

10. Usando la BD con Eloquent

Relaciones 1:1 (usuarios ↔ emails)

- En el **modelo** de la tabla maestra (**class Usuario**) añadimos este método:

```
public function email() {
    return $this->hasOne('App\Email');
}
```

- En el **modelo** de la tabla relacionada (**class Email**) añadimos este método:

```
public function usuario() {
    return $this->belongsTo('App\Usuario');
}
```

- Ya se puede recuperar el email a partir del usuario o a la inversa. Por ejemplo:

```
$email = Usuario::find(1)->email;
$user = Email::all()->first()->user;
```

10. Usando la BD con Eloquent

Relaciones 1:N (usuarios ↔ articulos)

- En el **modelo** de la tabla maestra (**class Usuario**) añadimos este método:

```
public function articulos() {
    return $this->hasMany('App\Articulo', 'idUsuario');
}
// ATENCIÓN: hemos tenido que indicar el nombre de la clave foránea
// (idUsuario) porque no habíamos respetado la convención de Laravel
// (usuario_id) al crear la tabla de artículos
```

- En el **modelo** de la tabla relacionada (**class Articulo**) añadimos este método:

```
public function usuario() {
    return $this->belongsTo('App\Usuario');
}
```

- Ya se pueden recuperar los artículos a partir del usuario o a la inversa. Por ejemplo:

```
$articulos = Usuario::find(1)->articulos;
foreach ($articulos as $articulo) {
   // Procesar cada artículo
```

10. Usando la BD con Eloquent

Relaciones N:N (usuarios ↔ roles)

- En el **modelo** de una de las tablas (**class Usuario**) añadimos este método:

```
public function roles() {
    return $this->belongsToMany('App\Rol');
}
```

- En el **modelo** de la otra tabla (**class Rol**) añadimos este método:

```
public function usuarios() {
    return $this->belongsToMany('App\Usuario');
}
```

- Ya se pueden recuperar los roles a partir del usuario o a la inversa. Por ejemplo:

```
$roles = Usuario::find(1)->roles;
foreach ($roles as $rol) {
   // Procesar cada rol
}
```

10. Usando la BD con Eloquent

Problemas frecuentes en relaciones N:N

- Eloquent supondrá que el nombre de la tabla de la relación se ha formado con los nombres de las dos tablas maestras en snake case y ordenadas alfabéticamente.
- Por ejemplo, en la relación N:N entre “usuarios” y “roles”, Eloquent supondrá que existe una tabla llamada “roles_usuarios”. Si no es así, la relación fallará.
- Se puede indicar otro nombre de tabla al definir la relación. Por ejemplo, en el modelo de usuarios (**class Usuario)**:

```
public function roles() {
    return $this->belongsToMany('App\Rol', 'usuarios_roles');
}
```

- También se pueden indicar los nombres de las claves foráneas si no siguen la convención de Laravel (usuario_id, rol_id, etc)

```
public function roles() {
    return $this->belongsToMany('App\Rol', 'usuarios_roles',
                                'id_usuario', 'id_rol');
}
```

10. Usando la BD con Eloquent

¿Usar atributos o métodos?

- ¿Te has fijado en que hemos creado un método para acceder a la tabla relacionada, pero estamos usando un atributo en su lugar?

```
public function articulos() {
    return $this->hasMany('App\Articulo');
}
public function loQueSea() {
    $arts = Usuario::find(1)->articulos;  // articulos, no articulos()
```

- El **atributo articulo** es un “atributo virtual” creado por Eloquent.

- Pero el **método articulos()** también existe, y puede usarse como una consulta, extendiéndola como necesitemos. Por ejemplo:

```
$arts = Usuario::find(1)->articulos()->where('titulo','foo')->first();
```

- **Más info en: https://laravel.com/docs/5.x/eloquent-relationships**

11. Usando la BD con QueryBuilder

11. Usando la BD con QueryBuilder

¿Qué es QueryBuilder?

- Eloquent permite usar la BD de forma simple y elegante en la mayor parte de las circunstancias.
- Aún así, puede haber situaciones en las que queramos un acceso de más bajo nivel a la BD. Para eso existe QueryBuilder.
- Ejemplos de uso:

```
$users = DB::table("users")->get();
$users = DB::table("users")->where("name", "=", "Ana")->first();
$users = DB::table("users")->where("edad", ">=", 18)->orderBy("apellidos");
$maxId = DB::table("users")->max("id");
$existe = DB::table("users")->where("id", "=", $id)→exists();
$users = DB::table("users")->select("nombre, apellidos as apell")->get();
```

- `$users = DB::table("users")->join("articles", "users.id", "=", "articles.id_user")->get();`
- Incluso es posible usar SQL estándar sin anestesia ni nada si así lo deseamos:

```
$result = DB::raw("SELECT * FROM users WHERE ...etc...");
```

12. Sesiones en Laravel

Sesiones en Laravel

- Las sesiones de configuran en /config/sessions.php
- El driver por defecto es *files.*
- En producción se recomienda *memcached* o *redis* (más rápidos). Son *daemons* del sistema programados para esta tarea.
- Para más seguridad, puede usarse el driver *database*.
- Laravel maneja dos tipos de variable según su persistencia:
  1) **Variables flash**: solo duran una petición y luego se autodestruyen.
  2) **Variables de sesión** convencionales: existen hasta que las destruimos expresamente.

12. Sesiones en Laravel

Sesiones en Laravel: variables flash

- Solo duran una petición y luego se autodestruyen.
- Se usan típicamente para enviar *feedback* al usuario.
- Por ejemplo:

  En el controlador:

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

  En la vista:

```
@if (session('mensaje'))
    {{ session('mensaje'); }
@endif
```

12. Sesiones en Laravel

Sesiones en Laravel: autenticación

```
$ php artisan make:auth
```

- Esto crea automáticamente las siguientes rutas en /routes/web.php:
  - **Routes::get("/login")** → Para mostrar el formulario de login
  - **Routes::post("/login")** → Para procesar el formulario de login
  - **Routes::post("/logout")** → Para cerrar la sesión
  - **Routes::get("/register")** → Para mostrar el fomulario de registro
  - **Routes::post("/register")** → Para procesar el formulario de registro
- También se crean los controladores LoginController y RegisterContoller: están en App/Http/Controllers/Auth.
- Se crean las vistas auth/login.blade.php, register.blade.php y layouts/app.blade.php (la plantilla que usarán login y register)
- Por último, se crea una vista de ejemplo (home.blade.php), un controlador de ejemplo (HomeController.php) y una ruta (/home), para mostrarnos cómo proteger partes de la aplicación con autenticación.

¡Y listo! Solo nos queda adaptar estas vistas y controladores a nuestras necesidades.

Sesiones en Laravel: variables de sesión

Las variables de sesión convencionales se manejan con la clase Session:

- **put()** almacena una variable de sesión:

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

- **push()** elimina una variable de sesión:

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

- **get()** devuelve el valor de una variable de sesión:

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

- **all()** devuelve todas las variables de sesión en un array:

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

- **flush()** elimina todas las variables de sesión:

```
Session::flush();
```

- **flash()** crea manualmente una variable de sesión de tipo *flash:*

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

Sesiones en Laravel: vistas

En las vistas, tenemos un par de directivas de Blade muy útiles relacionadas con las sesiones.

```
@auth
...   // Este código se ejecuta si existe un usuario logueado
@endauth
@guest
...   // Este código se ejecuta si NO existe usuario logueado
@endguest
```

Además, podemos acceder a los datos del usuario mediante el helper auth():

- **`auth()->user()`** → Devuelve el usuario actualmente logueado o null si no hay ninguna sesión abierta.

Sesiones en Laravel: middlewares

- Los **middlewares** son componentes software que capturan y filtran todas las peticiones HTTP que llegan a la aplicación.
- Están ubicados en App/Http/Middleware.
- Hay dos middlewares relacionados con la autenticación en Laravel: Authenticate (alias "auth") y RedirectIfAuthenticated (alias "guest"). Los alias se definen en App/Http/Kernel.php
- Podemos usar estos middlewares en el constructor de nuestros controladores para protegerlos en todo o en parte:

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

13. Helpers en Laravel

13. Helpers en Laravel

Qué son los helpers

- Un **helper** es un componente del framework diseñado para facilitar alguna tarea típica en el desarrollo de una aplicación web.
- Por ejemplo: el helper `url('ruta')` genera una ruta absoluta para referenciar cualquier componente de la aplicación:

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

  Generará este código:

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

  Eso permite que la ruta sea correcta en cualquier servidor, sin necesidad de modificar el código fuente.
- El uso de los helpers es optativo: el programador/a debe decidir si le resultan útiles o no.
- En Laravel 5.x desaparecieron muchos helpers que existían en Laravel 4.x para matener el framework lo más sencillo posible.
- Puedes encontrar una lista completa de helpers en: https://laravel.com/docs/5.x/helpers

13. Helpers en Laravel

Algunos helpers muy útiles: url

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

Generará este código:

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

Eso permite que la ruta sea correcta en cualquier servidor, sin necesidad de modificar el código fuente.

13. Helpers en Laravel

Algunos helpers muy útiles: route

- Es parecido a `url()`, pero sirve para routas con nombre en el enrutador.

- Por ejemplo, si en el enrutador tenemos una ruta como esta:

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

- ...podemos referirnos a ella como:

```
<a href='{{ url('mi-ruta') }}>Texto</a>
```

- ...o como:

```
<a href='{{ route('nombre-ruta') }}>Texto</a>
```

- **La segunda forma es la mejor**: permite cambiar la dirección que ve el usuario sin modificar el código fuente.

13. Helpers en Laravel

Algunos helpers muy útiles: request

Proporciona acceso a información sobre la petición (GET, POST o la que sea) con la que se cargó la página:

- **`request()->url()`** → Devuelve un string con la ruta actual (completa).
- **`request()->path()`** → Devuelve un string con la ruta actual (solo desde la raíz de la aplicación, sin http ni el nombre del servidor).
- **`request()->is("ruta")`** → Devuelve true si "ruta" coincide con la ruta actual. Admite wildcards (símbolos * y ?).
- **`request()->input("campo")`** → Devuelve el valor de "campo" (enviado desde formulario).
- **`request()->all()`** → Devuelve un array con todos los campos.
- **`request()->has("campo")`** → Devuelve true si en la petición existe un campo con el nombre indicado.
- **`request()->isMethod("método")`** → Devuelve true si la petición se hizo por el método indicado (POST, GET, PUT, etc).

El helper Request puede usarse en las vistas (como request()->url(), por ejemplo) o inyectarse en las funciones del controlador como una variable, así:

- **`public function mi-función(Request $request)`**

13. Helpers en Laravel

Algunos helpers muy útiles: redirect

- Muy útil cuando queremos redirigir al usuario hacia otra URL o acción.
- (Por ejemplo, para evitar que al pulsar F5 se reenvíen los datos de un formulario)
- Admite varias formas:

```
return redirect('user/login');
return redirect()->action('LoginController@login');
return back();
```

13. Helpers en Laravel

Algunos helpers muy útiles: auth

- Como vimos en la sección de sesiones y autenticación, este helper permite saber si existe algún usuario autenticado en la aplicación.
- **`auth()->user()`** devuelve el usuario autenticado (como un objeto) o null si nadie ha hecho login.
- Con el objeto User podemos acceder a todos los datos del usuario:

```
$user = auth()->user();
Bienvenido/a, {{ $user->name }}.
Este es su emial: {{ $user->email }}
```

13. Helpers en Laravel

Algunos helpers muy útiles: errors

- Se utiliza para conocer y mostrar los errores ocurridos en la validación de un formulario (v. apartado de validación de formularios).
- La variable $errors está disponible en todas las vistas gracias a que un Middleware (ShareErrorsFromSession) la inyecta automáticamente.
- Algunos métodos útiles:
- `$errors->all()` devuelve un array con todos los errores detectados.
- `$errors->any()` devuelve true si se ha detectado algún error.
- `$errors->first("campo")` devuelve el primer error de todos los que puedan afectar al campo indicado.

Contenido

14. Flujo de trabajo con Laravel

14. Flujo de trabajo con Laravel

Un posible flujo de trabajo con Laravel:

1) Instalar y configurar nueva aplicación.
2) Crear modelos (se supone que ya tendré la BD diseñada).
3) Crear migraciones y seeders. Lanzarlos para crear y poblar la BD.
4) Crear en el enrutador las entradas de la funcionalidad que voy a programar.
5) Crear el controlador (si no existe) para la funcionalidad que voy a programar.
6) Crear las funciones del controlador necesarias.
7) Crear las funciones del modelo necesarias (si no existen ya).
8) Crear las vistas necesarias.
9) Probar.
10) Repetir los pasos 4-9 para cada funcionalidad adicional.

Contenido

15. Aspectos avanzados

15. Aspectos avanzados (para investigar)

1) **Traducciones.**
2) **Paginación de resultados en vistas.**
3) **Páginas de error personalizadas.**
4) ***Routes model bindings* para URLs amigables.**
5) **Validación avanzada de formularios con *form requests*.**
6) **Optimizar el enrutador en servidores REST: Route::resource()**
7) Enviar emails con Laravel. Mailtrap. Sparkpost.
8) Laravel Mix para usar SASS, LESS o STYLUS con CSS y Javascript para el diseño.
9) Filtrar las rutas con expresiones regulares.
10) Crear helpers y middlewares.
11) Políticas de acceso complejas.
12) Collections.
13) Relaciones polimórficas con Eloquent.
14) Eventos y Listeners.
15) Actualizar una aplicación existente a una versión de Laravel posterior.
16) Memoria caché y mejora del rendimiento.
17) Decoradores e interfaces.
18) Pruebas: unit tests, integration tests, functional tests, acceptance tests. Mockery y Selenium.

15. Aspectos avanzados

Más info:

1) Referencia oficial de Laravel:
   https://laravel.com/docs/5.x (sustituye “x” por la versión que desees)
2) Buenas prácticas de programación con Laravel:
   http://www.laravelbestpractices.com
3) Screencasts (vídeos) sobre Laravel, PHP, JS y mucho más:
   https://laracasts.com/