

1 LARAVEL Framework - Tutorial 02 - Ampliación de conceptos en Laravel

1.1 Sumario

- 1 Rutas
 - ◆ 1.1 Parámetros en las rutas
 - ◇ 1.1.1 Rutas básicas
 - ◇ 1.1.2 Ruta que responde a cualquier tipo de petición HTTP
 - ◇ 1.1.3 Parámetros Obligatorios en las Rutas
 - ◇ 1.1.4 Parámetros Opcionales en las Rutas
- 2 Vistas en Laravel
 - ◆ 2.1 Pasando variables a las Vistas
 - ◆ 2.2 Plantillas Blade en Laravel
 - ◆ 2.3 Condicionales, bucles, sub-vistas, etc en plantillas Blade de Laravel
 - ◇ 2.3.1 Condicionales
 - ◇ 2.3.2 Bucles
 - ◇ 2.3.3 Sub-Vistas
 - ◇ 2.3.4 Mostrando textos de idioma
 - ◆ 2.4 Extender las plantillas Blade en Laravel
 - ◆ 2.5 HTMLBuilder y FormsBuilder en Laravel 5
- 3 Bases de datos en Laravel
 - ◆ 3.1 Conexión a bases de datos con Laravel
 - ◆ 3.2 SQL básico de consultas, inserciones, actualizaciones y borrados con Laravel
 - ◆ 3.3 Query Builder en Laravel
- 4 MVC y Teoría REST
 - ◆ 4.1 MVC
 - ◆ 4.2 REST (Representational State Transfer)
- 5 Controladores
 - ◆ 5.1 Controladores Básicos
 - ◆ 5.2 Controladores Implícitos
 - ◆ 5.3 Controladores RESTful Resource
- 6 Bases de datos avanzadas
 - ◆ 6.1 Migrations
 - ◆ 6.2 Schema Builder
 - ◆ 6.3 Seeding
 - ◆ 6.4 Modelos con Eloquent
 - ◆ 6.5 Relaciones
- 7 Miscelánea
 - ◆ 7.1 Middleware
 - ◆ 7.2 Validación de formularios
 - ◇ 7.2.1 Localización de los mensajes de error al Español (recomendado)
 - ◆ 7.3 Servidor web con php Artisan, modo mantenimiento, errores, etc. en Laravel 5
 - ◇ 7.3.1 Prueba de proyectos con servidor web desde Artisan
 - ◆ 7.4 Modo de Mantenimiento
 - ◆ 7.5 Gestionar errores 404
 - ◆ 7.6 Seeding con Faker
 - ◆ 7.7 Crear Migrations con definición de campos con PHP Artisan
 - ◆ 7.8 Permitir peticiones AJAX CORS (Cross-Origin Http Request)
 - ◆ 7.9 Configuración servidor de correo GMAIL en Laravel
 - ◆ 7.10 Ampliación de tiempo máximo token CSRF en Laravel

2 Rutas

2.1 Parámetros en las rutas

2.1.1 Rutas básicas

- Documentación Oficial sobre Routing: <http://laravel.com/docs/5.0/routing>

```
Route::get('/', function()
{
    return 'Hello World';
});

Route::post('foo/bar', function()
{
    return 'Hello World';
});

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

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

2.1.2 Ruta que responde a cualquier tipo de petición HTTP

```
Route::any('foo', function()
{
    return 'Hello World';
});
```

2.1.3 Parámetros Obligatorios en las Rutas

```
Route::get('batidos/{sabor}', function($sabor)
{
    return "Los batidos de $sabor está muy bien";
});
```

2.1.4 Parámetros Opcionales en las Rutas

```
// Se puede asignar un valor por defecto a ese parámetro opcional.

Route::get('batidos/{sabor?}', function($sabor='fresa')
{
    return "Los batidos de $sabor están muy bien";
});

Route::get('noticias/{titulo?}', function($titulo=null)
{
    if ($titulo==null)
    {
        return 'Listado de todos los artículos...';
    }

    return 'Contenido del artículo con título: '.$titulo;
});

Route::get('opcion/{param1}/b/{param2?}', function($p1,$p2='default')
{
    return 'Parámetro 1 = '.$p1.' y parámetro 2='.$p2;
});
```

3 Vistas en Laravel

- Documentación Oficial sobre Vistas: <http://laravel.com/docs/5.0/views>
- Las vistas se encuentran en **resources/views**.

- Ejemplo de una vista almacenada en **resources/views/greeting.php**.

```
<html>
  <body>
    <h1>Hello, <?php echo $name; ?></h1>
  </body>
</html>
```

- La vista se devuelve al navegador de la siguiente forma:

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

3.1 Pasando variables a las Vistas

- Fichero de rutas: **app/Http/routes.php**:

```
<?php

/*
|-----
| Application Routes
|-----
|
| Here is where you can register all of the routes for an application.
| It's a breeze. Simply tell Laravel the URIs it should respond to
| and give it the controller to call when that URI is requested.
|
*/

Route::get('/',function()
{
    // 2 formas de pasar variables a la vista.
    $animal1='Leon';
    $animal2='Cebra';
    $capital1='Madrid';
    $capital2='París';

    $lista=array($animal1,$animal2,$capital1,$capital2);

    // Opción 1: pasando 1 variable.
    // return View('hola')->with('variable',$animal1);

    // Opción 2: Pasando un array con una variable.
    // return View('hola',array('variable'=>$animal2));

    // Opción 3: pasando más de una variable.
    //return View('hola')->with('variable',$animal1)->with('capital',$capital1);

    // Opción 4: Pasando en un array más de una variable.
    // return View('hola',array('variable'=>$animal2,'capital'=>$capital2));

    // Opción 5: Métodos Mágicos withVar (with+Nombre de variable a pasar)
    // return View('hola')->withAnimal1('Gato montés');

    // Opción 6: Enviando una lista de variables.
    return View('hola')->with('lista',$lista);

    // También se podrían pasar parámetros de la ruta a la vista

Route::get('pruebas/{algo?}',function($algo='')
{
    return View('hola')->with('variable',$algo);
});

});
```

- **Fichero de vista: resources/views/hola.blade.php:**

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title>Ejemplo de Vista en Laravel</title>
</head>
<body>
<h1>Bienvenido a Laravel</h1>
Se puede escribir texto PHP abriendo código de PHP.<br/>
Con Blade en lugar de abrir código PHP se puede hacer algo como:<br/>
<h2>{{ $variable }}</h2>
<h3>{{ $capital }}</h3>

<ul>
<li>{{ $lista[0] }}</li>
<li>{{ $lista[1] }}</li>
<li>{{ $lista[2] }}</li>
<li>{{ $lista[3] }}</li>
</ul>
</body>
</html>
```

3.2 Plantillas Blade en Laravel

- **Documentación Oficial sobre Plantillas en Laravel:** <http://laravel.com/docs/5.0/templates>
- Se almacenan en **resources/views**.
- Las vistas tienen la extensión **.blade.php**.
- **Ejemplo de Ruta que renderiza una plantilla Blade:**

```
Route::get('/',function()
{
return View('hola');
});
```

- **Ejemplo de plantilla Blade resources/views/hola.blade.php:**

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title>Ejemplo de Vista en Laravel</title>
</head>
<body>
<h1>Bienvenido a Laravel</h1>
Se puede escribir texto PHP abriendo código de PHP.<br/>
</body>
</html>
```

3.3 Condicionales,bucles,sub-vistas,etc en plantillas Blade de Laravel

3.3.1 Condicionales

- **Ejemplo de plantilla Blade que incluye Condicionales:**

```
/*
    En app/Http/routes.php tenemos algo como:
    $amigos=array('luis','marta','pedro');
    return View('hola')->withAmigos($amigos);
*/

// Código de la plantilla hola.blade.php:

@if (count($amigos) ==1)
Tengo un amigo llamado {{$amigos[0]}}
```

```

@elseif (count($amigos)>1)
Tengo múltiples amigos.
@else
Soy muy solitario.
@endif

```

3.3.2 Bucles

- Ejemplo de plantilla Blade que incluye **Bucles**:

```

/*
    En app/Http/routes.php tenemos algo como:
    $amigos=array('luis','marta','pedro');
    return View('hola')->withAmigos($amigos);
*/

// Código de la plantilla hola.blade.php con un Bucle normal:

@for($i=0; $i<count($amigos); $i++)
<li>{{$amigos[$i]}}</li>
@endfor

// Código de la plantilla hola.blade.php con un Bucle foreach:

@foreach($amigos as $amigo)
<li>{{$amigo}}</li>
@endforeach

// Ejemplo de bucle while
@while (true)
    <p>I'm looping forever.</p>
@endwhile

```

3.3.3 Sub-Vistas

Se incluyen con:

```
@include('nombreVista')
```

3.3.4 Mostrando textos de idioma

Con Laravel se puede trabajar con multi-idioma y Blade nos permite definir las cadenas de idioma a mostrar en nuestra aplicación.

```

# Suponiendo que tengamos una clave llamada welcome en el fichero messages.php, dentro de la carpeta de idioma correspondiente.

# Podremos mostrar en Blade el texto con

@lang('messages.welcome')

# Más información sobre Localización en:
https://laravel.com/docs/5.1/localization

```

3.4 Extender las plantillas Blade en Laravel

- Documentación sobre Blade Templating: <http://laravel.com/docs/5.0/templates>

Supongamos que queremos hacer una **mini aplicación con 3 opciones de menú**:

- Código fuente de **app/Http/routes.php**:

```

<?php

/*
|-----
| Application Routes
|-----
|

```

```
| Here is where you can register all of the routes for an application.
| It's a breeze. Simply tell Laravel the URIs it should respond to
| and give it the controller to call when that URI is requested.
```

```
|
*/
```

```
Route::get('/',function()
{
return View('inicio');
});
```

```
Route::get('about',function()
{
return View('about');
});
```

```
Route::get('contactar',function()
{
return View('contactar');
});
```

• Código fuente de **resources/views/inicio.blade.php**:

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title>Inicio</title>
</head>
<body>
<nav>
<a href=".">Inicio</a>
<a href="about">About</a>
<a href="contactar">Contactar</a>
</nav>

<section>Bienvenido al Inicio de la página web de esta aplicación.</section>
</body>
</html>
```

Código fuente de **resources/views/about.blade.php**:

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title>Inicio</title>
</head>
<body>
<nav>
<a href=".">Inicio</a>
<a href="contactar">Contactar</a>
</nav>

<section>Sobre nosotros.</section>
</body>
</html>
```

• Código fuente de **resources/views/contactar.blade.php**:

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title>Inicio</title>
</head>
<body>
<nav>
<a href=".">Inicio</a>
<a href="about">About</a>
</nav>

```

```
<section>Contacte con nosotros</section>
</body>
</html>
```

- Si nos fijamos vemos que **las tres Vistas comparten prácticamente el mismo código** salvo pequeñas diferencias.
 - Entonces podemos crear un fichero común esqueleto y programar ahí dentro todo el contenido de las vistas:
 - Haremos una nueva carpeta dentro de **resources/views**, a la que llamaremos **layouts** y ahí dentro escribiremos el fichero **esqueleto** que llamaremos **master.blade.php**.
 - En el fichero **master.blade.php** usamos **@yield('punto-entrada')** para indicar qué queremos que se ponga en ese punto de entrada.
- Código fuente del fichero master **resources/views/layouts/master.blade.php**:

```
<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title>Inicio</title>
</head>
<body>
<nav>
<a href="/">Inicio</a> |
@yield('menu-navegacion')
</nav>



<section>
@yield('contenido')
</section>

</body>
</html>
```

- Ahora adaptaremos cada una de las vistas para extender su uso a la plantilla esqueleto definida anteriormente en **layouts/master/master.blade.php**.
 - Se comienza al principio extendiendo la plantilla esqueleto con **@extends('layouts/master')** o también **@extends('layouts.master')**
 - Cada **punto-entrada** definido en la plantilla **master** con **@yield**, tendremos que crear en nuestra vista su correspondiente contenido con **@section('punto-entrada')** y **@stop**
 - Documentación sobre Blade Templating: <http://laravel.com/docs/5.0/templates>
- Código fuente del fichero master **resources/views/inicio.blade.php**:

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

@section('menu-navegacion')
<a href="about">About</a> |
<a href="contactar">Contactar</a>
@stop

@section('contenido')
Bienvenido al Inicio de la página web de esta aplicación.
@stop
```

- Código fuente del fichero master **resources/views/about.blade.php**:

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

@section('menu-navegacion')
<a href="contactar">Contactar</a>
@stop

@section('contenido')
Sobre nosotros.
@stop
```

- Código fuente del fichero master **resources/views/contactar.blade.php**:

```

@extends('layouts.master')

@section('menu-navegacion')
<a href="about">About</a>
@stop

@section('contenido')
Contacte con nosotros
@stop

```

- Otra forma un poco más **profesional** de hacerlo consiste en crear **@section** en la plantilla master y terminar con **@show** en lugar de **@stop**.
- De esta forma **se sobrescribirá el contenido enviado en la vista en esa sección definida en master**.
- Si queremos que mantenga parte del contenido de la section definida en master y añada lo que se le envía desde la vista hay que añadir en cada una de las **vistas @parent** para que añada a master el contenido que se le envía desde la vista.
- Código fuente del fichero master **resources/views/layouts/master.blade.php**:

```

<!DOCTYPE html>
<html lang="es">
<head>
<meta charset="UTF-8">
<title>Inicio</title>
</head>
<body>
<nav>
@section('menu-navegacion')
<a href="/">Inicio</a> |
@show
</nav>



<section>
@yield('contenido')
</section>

</body>
</html>

```

- Código fuente del fichero master **resources/views/inicio.blade.php**:

```

@extends('layouts.master')

@section('menu-navegacion')
@parent
<a href="about">About</a> |
<a href="contactar">Contactar</a>
@stop

@section('contenido')
Bienvenido al Inicio de la página web de esta aplicación.
@stop

```

- Código fuente del fichero master **resources/views/about.blade.php**:

```

@extends('layouts.master')

@section('menu-navegacion')
@parent
<a href="contactar">Contactar</a>
@stop

@section('contenido')
Sobre nosotros.
@stop

```

- Código fuente del fichero master **resources/views/contactar.blade.php**:

```

@extends('layouts.master')

```



```

@section('menu-navegacion')
@parent
<a href="about">About</a>
@stop

@section('contenido')
Contacte con nosotros
@stop

```

3.5 HTMLBuilder y FormsBuilder en Laravel 5

- En Laravel 5 se han eliminado HTMLBuilder y FormsBuilder.
- Si se desean crear formularios o generar contenido HTML rápidamente es necesario instalar esos módulos.
- Información de la instalación y uso de dichos módulos en: <http://laravelcollective.com/>

4 Bases de datos en Laravel

4.1 Conexión a bases de datos con Laravel

- Laravel nos permite utilizar múltiples sistemas de bases de datos: sqlite, MySQL, Postgress y SQL Server.
- Por defecto está configurado para usar mysql. Véase el fichero `app/config/database.php` en la línea 29: `'default' => 'mysql'` .

- Para usar bases de datos primero tendremos que crearlas desde PHPMYAdmin o línea de comandos.
- A continuación tendremos que editar el fichero de entorno `.env` y adaptar los siguientes parámetros a nuestra configuración.
- Código fuente del fichero de entorno `.env`:

```

DB_HOST=localhost
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret

```

4.2 SQL básico de consultas, inserciones, actualizaciones y borrados con Laravel

- Si queremos realizar las típicas operaciones CRUD usando sentencias SQL podemos hacerlo empleando métodos de la clase **DB**.
- Ejemplos de uso de operaciones en bases de datos en una ruta por defecto:

```

Route::get('/',function()
{
// Consultar un único usuario.
// Devuelve un array cuya posición 0 contiene un objeto con la info. de ese usuario.
$user=DB::select("select * from users where id=2");

// Mostramos la variable y terminamos la ejecución. dd($variable)
dd($user);

// Si no queremos que devuelva un array entonces usaremos selectOne
// Devolverá un objeto con el contenido de ese registro.
$user=DB::selectOne("select * from users where id=2");
return $user->nombre. ' tiene como apellido '.$user->apellidos;

// Consultar todos los usuarios.
$users=DB::select("select * from users");
dd($users);

// Insertar un nuevo usuario:
DB::insert("INSERT into users(nombre,apellidos,edad) VALUES(?,?,?)",array('Marta','García Pérez',43));
return "Guardado correctamente.";

// Actualizar un usuario existente:
DB::update("UPDATE users set nombre=? where id=?",array('Martita',3));
return "Actualizado correctamente.";

// Borrar un usuario existente:
DB::delete("DELETE FROM users where id=?",array(3));

```

```
return "Borrado correctamente.";
});
```

4.3 Query Builder en Laravel

- **Documentación Oficial de Query Builder en Laravel:** <http://laravel.com/docs/5.0/queries>
- Usando Query Builder nos olvidamos de usar SQL específico de cada base de datos y Laravel se encarga de la traducción al driver correspondiente.
- Ejemplo de código dónde se usa **Query Builder** de Laravel:

```
Route::get('/',function()
{
// Documentación oficial de Query Builder en: http://laravel.com/docs/5.0/queries

// Consulta usando QueryBuilder.
// Select * from users;
$users=DB::table('users')->get();
dd($users);

// Consulta de un único usuario.
// Select * from users where id=2;
// Devuelve un array que contiene el objeto con la información.
$user=DB::table('users')->where('id',2)->get();
dd($user);

// Consulta de un único usuario.
// Select * from users where id=2;
// Devuelve un objeto con la información.
$user=DB::table('users')->where('id',2)->first();
dd($user);

// Consulta con cláusula >
// Select * from users where id>3;
// Devuelve un array que contiene cada objeto con la información.
$users=DB::table('users')->where('id','>',3)->get();
dd($users);

// Consulta con métodos Mágicos en las cláusulas where.
// Select * from users where Edad=34;
// Devuelve un objeto con la información.
$user = DB::table('users')->whereEdad(34)->first();
dd($user);

// Insertar datos en la tabla.
DB::table('users')->insert(array('nombre'=>'Carlos','apellidos'=>'Díaz Blanco','edad'=>53));
return("Insertado correctamente");

// Se puede utilizar insertGetId( en lugar de insert) para obtener el ID recién insertado.
$id= DB::table('users')->insertGetId(array('nombre'=>'Carlos','apellidos'=>'Díaz Blanco','edad'=>53));
return("Insertado correctamente con el ID: $id");

// Actualizar datos en la tabla.
DB::table('users')->where('id',1)->update(array('nombre'=>'Carlitos','apellidos'=>'xxx','edad'=>24));
return("Actualizado correctamente");

// Actualizar datos con métodos Mágicos:
DB::table('users')->whereId(1)->update(array('nombre'=>'Carlitos','apellidos'=>'xxx','edad'=>25));
return("Actualizado correctamente");

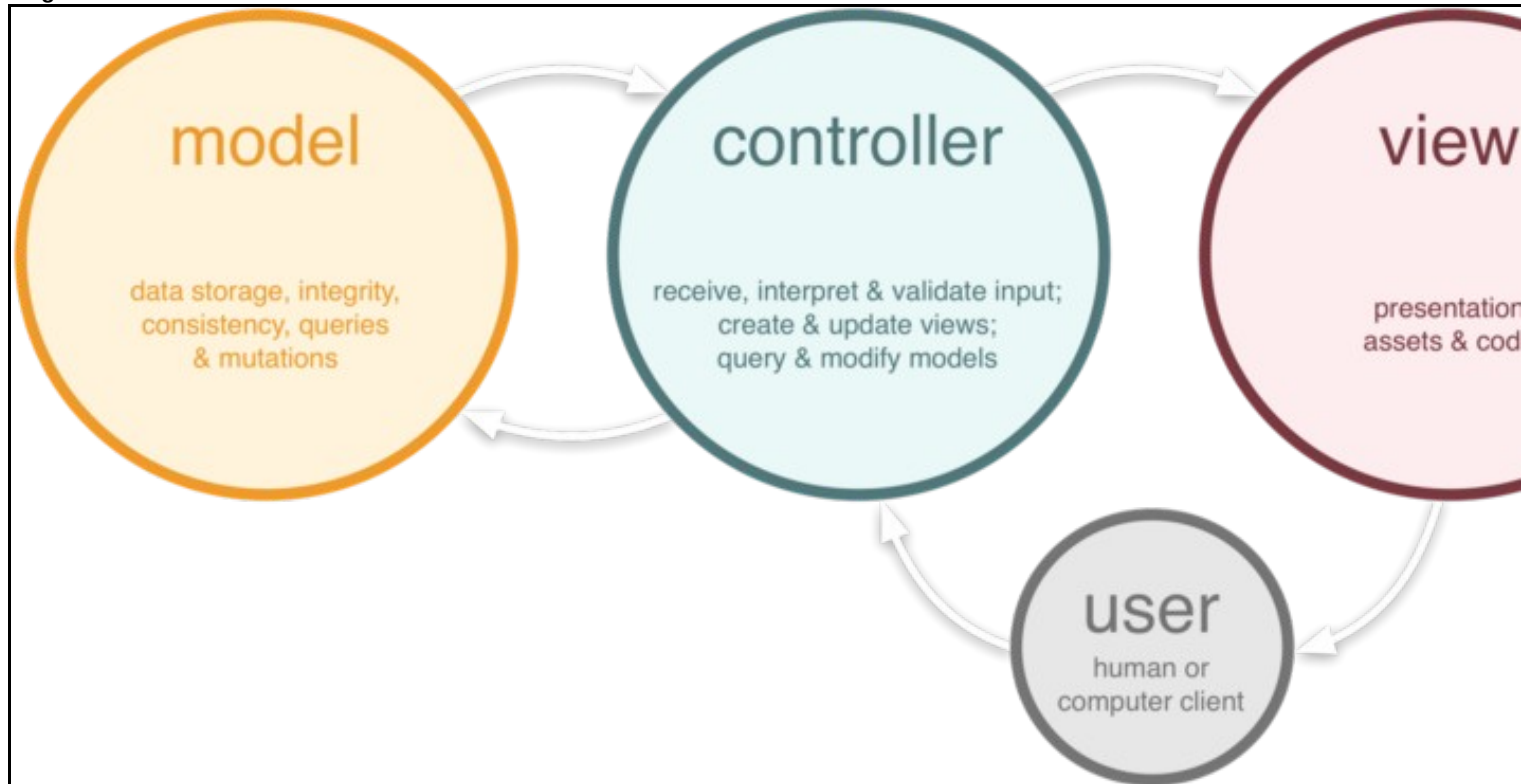
// Borrar datos en la tabla.
DB::table('users')->where('apellidos','xxx')->delete();
return("Borrado correctamente");

// Borrar datos con métodos Mágicos:
DB::table('users')->whereApellidos('Galvez')->delete();
return("Borrado correctamente");
});
```

5 MVC y Teoría REST

5.1 MVC

Diagrama del Modelo Vista Controlador en Laravel:



- **Modelos** se definen en: **app**
- **Vistas** se definen en: **resources/views**
- **Controladores** se definen en: **app/Http/Controllers**
- **Rutas** se definen en: **app/Http/routes.php**

5.2 REST (Representational State Transfer)

Un recurso es una representación abstracta de una clase particular de objeto que gestiona habitualmente una aplicación web.

Ejemplos:

- Receta (en una web de cocina)
- Usuario (en una web social)
- Libro (en una web librería)

Para cada uno de esos recursos tendremos que realizar un mantenimiento como por ejemplo:

- Ver todas las recetas **/recetas**
- Ver una receta específica por ejemplo de macarrones **/recetas/macarrones**
- Crear una nueva receta **/recetas/create** [necesario un **formulario**]
- Actualizar/borrar una receta existente **/recetas/macarrones/edit** [necesario un **formulario**]

Se utilizarán **estos verbos HTTP para solicitar acciones a la API RESTful**.

HTTP Verbs used in RESTful API

GET

Retrieves a resource
Guaranteed not to cause side-effects (SAFE)
Results are cacheable

POST

Creates a new resource (process state)
Unsafe: effect of this verb isn't defined by HTTP

PUT

Updates an existing resource
Used for resource creation when client knows URI
Can call N times, same thing will always happen (idempotent)

PATCH

Updates an existing resource (partially)
Can call N times, same thing will always happen (idempotent)

DELETE

Removes a resource
Can call N times, same thing will always happen (idempotent)

- GET -> Retrieve
- POST -> Create
- PUT -> Update
- PATCH -> Update
- DELETE -> Destroy

6 Controladores

En lugar de definir todas las rutas de la lógica de nuestra aplicación en **app/Http/routes.php**, podemos organizar toda esa lógica y comportamiento usando **clases Controller**.

```
// Ejemplo de rutas en app/Http/routes.php

Route::get('/hello',function()
{
return View('hello');
});

Route::get('/',function()
{
return View('inicio');
});

Route::post('/hello',function()
{
return "Almacenando datos en esta ruta";
});
```

6.1 Controladores Básicos

- **Documentación Oficial sobre controladores Básicos:** <http://laravel.com/docs/5.0/controllers#basic-controllers>
- Los controladores pueden agrupar peticiones HTTP cuya lógica es similar en una clase.

- Los controladores se definen y guardan en **app/Http/Controllers**.

```
// Ejemplo de código de un controlador:
<?php namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

class UserController extends Controller {

    /**
     * Show the profile for the given user.
     *
     * @param int $id
     * @return Response
     */
    public function showProfile($id)
    {
        return view('user.profile', ['user' => User::findOrFail($id)]);
    }

}
```

- Llamada del método showProfile desde una ruta:

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

6.2 Controladores Implícitos

- Laravel permite crear fácilmente una ruta que gestionará todo las sub-rutas que cuelguen de esa ruta principal.
- Se hace definiendo la ruta del tipo **Route::controller**:

```
// app/Http/routes.php
// El método Route::controller acepta 2 argumentos. El primero es la URI Base que el controlador gestiona y el segundo es el nombre

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

- Sólo nos queda crear los métodos en el Controlador. Por ejemplo:

```
// app/Http/Controllers/UserController.php

class UserController extends BaseController {

    public function getIndex()
    {
        // Equivale a acceder a /users/ usando GET
    }

    public function postProfile()
    {
        // Cuando accedemos a /users/Profile usando POST
    }

    public function getAdminProfile()
    {
        // Cuando accedemos a /users/admin-profile usando método GET.
    }

}
```

6.3 Controladores RESTful Resource

- Documentación Oficial sobre RESTful Resource Controllers <http://laravel.com/docs/5.0/controllers#restful-resource-controllers>
- Este tipo de controladores resource nos permiten definir rutas RESTful muy fácilmente.
- Para crear este tipo de controladores podemos usar Artisan:

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

- En `app/Http/routes.php` definimos la llamada a un controlador **Resource**:

```
// app/Http/routes.php
Route::resource('photo', 'PhotoController');
```

- **Acciones gestionadas por defecto en un Controlador Resource:**

Verb	Path	Action	Route Name
GET	/resource	index	resource.index
GET	/resource/create	create	resource.create
POST	/resource	store	resource.store
GET	/resource/{photo}	show	resource.show
GET	/resource/{photo}/edit	edit	resource.edit
PUT/PATCH	/resource/{photo}	update	resource.update
DELETE	/resource/{photo}	destroy	resource.destroy

- **Ejemplo de un controlador Resource customizado dónde se especifica un subconjunto de acciones que gestionará esa ruta:**

```
// app/Http/routes.php

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

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

- *** Ejemplo de un controlador Resource Anidado:**

```
// app/Http/routes.php
// Si queremos hacer controladores resource anidados usaremos la notación de punto a la hora de definir el controlador.
Route::resource('photos.comments', 'PhotoCommentController');

// Esta ruta registrará un recurso anidado que podrá ser accedido con URLs del estilo:
photos/{photos}/comments/{comments}
```

7 Bases de datos avanzadas

7.1 Migrations

- Las **migraciones** son un tipo de **control de versiones sobre la base de datos**.
- Permiten que un equipo pueda modificar el esquema y mantenerlo actualizado.
- Las migraciones están muy relacionadas con **[Schema Builder <http://laravel.com/docs/5.0/schema>]** (Clase que nos permite crear y modificar la estructura de tablas, etc..)
- **[Documentación Oficial sobre Migrations en Laravel <http://laravel.com/docs/5.0/migrations>]**
- La carpeta dónde se crean las Migrations es **database/migrations**.

- Para poder trabajar con Migrations hay que instalar previamente una **tabla** en nuestra base de datos llamada **migrations**. En Laravel 5 ya se instala de forma automática al ejecutar las Migrations si ésta no hubiese sido instalada antes. Si queremos instalarla manualmente haremos:

```
php artisan migrate:install
```

- Para **crear una plantilla de Migrations** se hará con PHP Artisan:

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

El parámetro `--table` y `--create` se puede utilizar para indicar el nombre de la tabla en la Migration y cuando la Migration debe crea

```
php artisan make:migration add_votes_to_users_table --table=users
```

```
php artisan make:migration create_users_table --create=users
```

- Para ejecutar las Migrations se hará con PHP Artisan:

```
php artisan migrate
```

- Para hacer un **Rollback** (deshacer lo último ejecutado):

```
php artisan migrate:rollback
```

```
# Si obtenemos un error al hacer un rollback ejecutar el siguiente comando y luego hacer el rollback:  
composer dump-autoload
```

- Para hacer un **Rollback de todas las Migrations**:

```
php artisan migrate:reset
```

- Para hacer un **Rollback de todas las Migrations y ejecutarlas de nuevo**:

```
php artisan migrate:refresh --seed
```

- Ejemplo de código de un fichero Migration:

```
<?php  
  
use Illuminate\Database\Schema\Blueprint;  
use Illuminate\Database\Migrations\Migration;  
  
class CrearTablaUsuarios extends Migration {  
  
    /**  
     * Run the migrations.  
     *  
     * @return void  
     */  
    public function up()  
    {  
        // Se podría definir una tabla con una sentencia SQL:  
        DB::statement("                CREATE TABLE usuarios(  
                                     id integer primary key auto_increment,  
                                     nombre varchar(255) unique not null,  
                                     edad integer  
                                     )  
                ");  
  
        // O bien se podría definir usando Schema Builder (recomendado)  
        Schema::create('usuarios', function(Blueprint $table)  
        {  
            $table->increments('id');  
            $table->string('nombre');  
            $table->integer('edad');  
  
            // Se añaden además los campos timestamps (created_at, updated_at):  
            $table->timestamps();  
        });  
    }  
  
    /**  
     * Reverse the migrations.  
     *  
     * @return void  
     */  
    public function down()  
    {  
        Schema::drop('usuarios');  
    }  
}
```

```
}
```

7.2 Schema Builder

- Usaremos **Schema Builder para definir de forma más sencilla sin usar comandos SQL**, las tablas, campos, claves, claves foráneas, etc... que se crearán dentro de las Migrations.
- Documentación Oficial sobre **Schema Builder en Laravel 5**

- **Ejemplo de Schema Builder** utilizado dentro de un fichero Migration en `database/migrations/2015_03_31_233224_vehiculos_migration.php`:

```
<?php

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

class VehiculosMigration extends Migration {

    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('vehiculos', function(Blueprint $table)
        {

            // Tipos de columnas en: http://laravel.com/docs/5.0/schema#adding-columns

            $table->increments('serie');
            $table->string('color');
            $table->float('cilindraje');
            $table->integer('potencia');
            $table->float('peso')->nullable();
            $table->integer('fabricante_id')->unsigned();
            $table->foreign('fabricante_id')->references('id')->on('fabricantes');
            $table->timestamps();
        });
    }

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

}
```

- Si quisiéramos **agregar un nuevo campo a la tabla** vehículos podríamos quedar una nueva migración y tendríamos que hacer algo como:

```
<?php

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

class AddCombustibleField extends Migration {

    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
```



```

Schema::table('vehiculos', function(Blueprint $table)
{
    $table->string('combustible');
});
}

/**
     * Reverse the migrations.
     *
     * @return void
     */
public function down()
{
    Schema::dropColumn(array('combustible'));
}

}

```

7.3 Seeding

- A través de un **Seeder** podremos insertar registros en las tablas que hemos creado con las Migrations.
- En el directorio **database/seeds** se encuentran todos los ficheros de Seeders.
- En el fichero **database/DatabaseSeeder.php** se definen las llamadas a todos los seeders que hemos programado.
- Ejemplo de fichero **database/DatabaseSeeder.php**:

```

<?php

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

use App\User;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        //Model::unguard();

        $this->call('FabricanteSeeder');
        $this->call('VehiculoSeeder');

        User::truncate();
        $this->call('UserSeeder');
    }
}

```

- Ejemplo de fichero **database/FabricanteSeeder.php** que usa **Faker** para generar datos aleatorios:

```

<?php

use Illuminate\Database\Seeder;
use App\Fabricante;
use Faker\Factory as Faker;

// Para poder usar Faker hay que instalarlo con Composer previamente:
// composer require fzaninotto/faker --dev

class FabricanteSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
}

```

```

public function run()
{
    $faker=Faker::create();

    for ($i=0;$i<3;$i++)
    {
        Fabricante::create
        ([
            'nombre'=>$faker->word(),
            'telefono'=>$faker->randomNumber(7)
        ]);
    }
}
}

```

7.4 Modelos con Eloquent

- Documentación Oficial sobre Eloquent ORM en: <http://laravel.com/docs/5.0/eloquent>
- Los modelos en Laravel 5 se encuentran en la carpeta **app**.
- Utilizamos **Eloquent ORM** para crear el modelo.

- Cada tabla es una clase.
- Dentro de esa clase definiremos todas las propiedades de la tabla, campos, relaciones, etc..

- Se puede crear una **plantilla modelo con PHP Artisan**:

```
php artisan make:model Fabricante
```

- Ejemplo de modelo **app/Fabricante.php**:

```

<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Fabricante extends Model
{
    protected $table="fabricantes";

    // Por seguridad para evitar actualizaciones masivas de los campos se indica cuales
    // son los campos permitidos sobre los que podemos hacer ese tipo de actualizaciones o inserciones.
    protected $fillable = array('nombre','telefono');

    // Aquí ponemos los campos que no queremos que se devuelvan en las consultas.
    protected $hidden = ['created_at','updated_at'];

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

}

```

- Si quisiéramos **insertar un registro en una tabla Fabricante**, por ejemplo dentro de una ruta de ejemplo (/insertar) podríamos hacer algo como:

```

use App\Fabricante;

Route::get('insertar',function()
{
    $fabricante= Fabricante::create(array(
        'nombre'=>'Mercadina',
        'telefono'=>9838393
    ));
    return $fabricante->id;
});

```

- Para **obtener todos los registros**:

```
$fabricante = Fabricante::all(); // Nos devolverá todos los registros de Fabricantes.
```

```
// Para obtener algunos registros:
$fabricante = Fabricante::where('id','<',12); // Fabricantes cuyo id < 12.
```

- Información completa sobre todos los métodos **insert**, **update**, **delete**, etc, en: <http://laravel.com/docs/5.0/eloquent>

7.5 Relaciones

- Para gestionar las relaciones entre las tablas en Laravel lo especificaremos en su Modelo.
- Más información sobre Relaciones en: <http://laravel.com/docs/5.0/eloquent#relationships>
- Ejemplo de modelo **app/Fabricante.php** dónde se hace la relación entre las dos tablas Fabricantes y Vehiculos:

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Fabricante extends Model
{
    protected $table="fabricantes";

    // Por seguridad para evitar actualizaciones masivas de los campos se indica cuales
    // son los campos permitidos sobre los que podemos hacer ese tipo de actualizaciones o inserciones.
    protected $fillable = array('nombre','telefono');

    // Aquí ponemos los campos que no queremos que se devuelvan en las consultas.
    protected $hidden = ['created_at','updated_at'];

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

}
```

- Ejemplo de modelo **app/Vehiculo.php** dónde se hace la relación entre las vehículo y Fabricante:

```
<?php namespace App;

use Illuminate\Database\Eloquent\Model;

class Vehiculo extends Model
{
    protected $table="vehiculos";
    protected $primaryKey='serie';
    protected $fillable = array('color','cilindraje','potencia','peso','fabricante_id');

    // Aquí ponemos los campos que no queremos que se devuelvan en las consultas.
    protected $hidden = ['created_at','updated_at'];

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

}
```

- En las Migrations usaremos la relación `->foreign('')` para indicar la relación con la otra tabla.
- Ejemplo de parte de código en una Migration **database/migrations/2015_03_31_233224_vehiculos_migration** que relaciona la tabla vehiculos con fabricantes:

```
Schema::create('vehiculos', function(Blueprint $table)
{
    $table->increments('serie');
    $table->string('color');
    $table->float('cilindraje');
    $table->integer('potencia');
    $table->float('peso');
    $table->integer('fabricante_id')->unsigned();
    $table->foreign('fabricante_id')->references('id')->on('fabricantes');
```

```
$stable->timestamps();
});
```

- Ejemplo de código en una ruta para **obtener todos los vehículos de un fabricante y también para mostrar el fabricante de un vehículo**:

```
Route::get('pruebas',function()
{
// Queremos saber los vehiculos del fabricante 2.
$vehiculos= Fabricante::find(2)->vehiculos()->get();

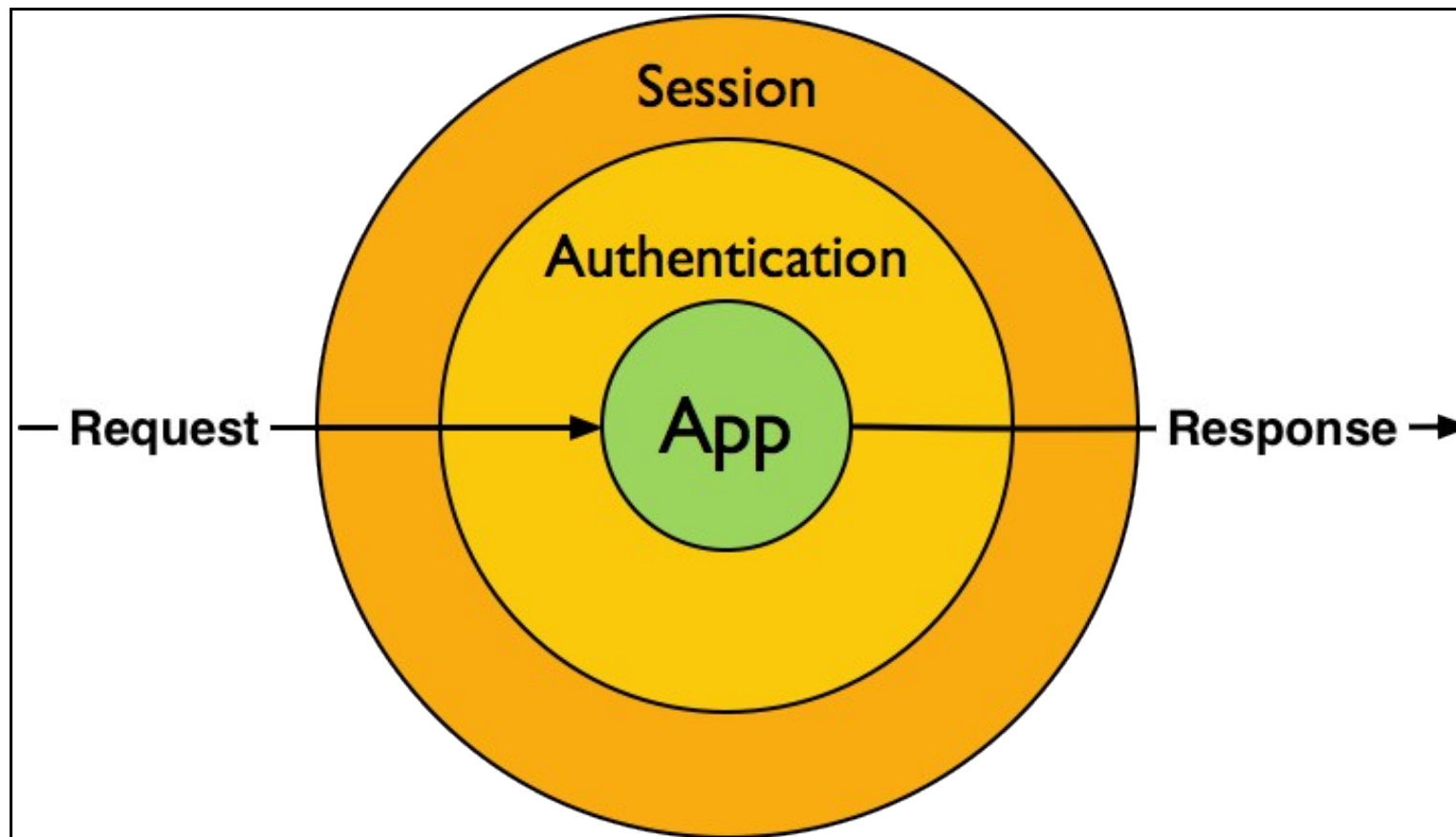
// O también otra forma:
// $vehiculos= Fabricante::find(2)->vehiculos;
return $vehiculos;

// Ejemplo a la inversa: obtener el fabricante de un vehiculo.
// Supongamos que tenemos un vehiculo y queremos saber su fabricante.
$fabricante= Vehiculo::find(3)->fabricante;
return $fabricante;
});
```

8 Miscelánea

8.1 Middleware

- **HTTP Middleware:** <http://laravel.com/docs/master/middleware>
- Los filtros nos permiten controlar el acceso a diferentes zonas de nuestra web dependiendo de si se cumplen o no las condiciones que definamos.
- **Middlewares proporcionan un mecanismo para filtrar las peticiones HTTP** que entran en nuestra aplicación. Por ejemplo, Laravel incluye un middleware que verifica si el usuario de la aplicación está autenticado.
- Los middleware se almacenan en la ruta **app/Http/Middleware**;
- Todo Middleware en Laravel 5 usa el componente Closure(función callback del middleware) y debe implementar la interfaz Middleware.



- Para crear un Middleware se puede hacer con PHP Artisan:

```
php artisan make:middleware NombreMiddleware
```

- Ejemplo de código de Middleware:

```
//.....  
public function handle($request, Closure $next)  
{  
    if (!$request->input('age') && $request->path() != "home")  
    {  
        return redirect('usuario/20');  
    }  
    return $next($request);  
}  
//.....
```

- Una vez creado el middleware hay que indicarle a Laravel dónde tiene que usarlo.

- ◆ Se puede registrar **globalmente** el Middleware en **app/Http/Kernel.php** en la propiedad **protected \$middleware**.
- ◆ Ejemplo de Middleware **registrado globalmente**:

```
/**  
 * The application's route middleware.  
 *  
 * @var array  
 */  
protected $routeMiddleware = [  
    'custom' => 'App\Http\Middleware\CustomMiddleware', //nuestro middleware  
    'auth' => 'App\Http\Middleware\Authenticate',  
    'auth.basic' => 'Illuminate\Auth\Middleware\AuthenticateWithBasicAuth',  
    'guest' => 'App\Http\Middleware\RedirectIfAuthenticated',  
];
```

- ◆ Middlewares **asignados en rutas**.
◆ Ejemplo de Middleware **registrado en una ruta**:

```
$router->get('foooo', [  
    'uses' => 'App\Http\Controllers\HomeController@foo',  
    'as' => NULL,  
    'middleware' => ['custom'],  
    'where' => [],  
    'domain' => NULL,  
]);
```

- Las llamadas a Middlewares se pueden especificar en las rutas o dentro del constructor de los controladores, por ejemplo:

```
////////////////////////////////////  
Route::get('profile', [  
    'middleware' => 'auth',  
    'uses' => 'UserController@showProfile'  
]);  
  
////////////////////////////////////  
  
class UserController extends Controller {  
  
    /**  
     * Instantiate a new UserController instance.  
     */  
    public function __construct()  
    {  
        $this->middleware('auth');  
  
        $this->middleware('log', ['only' => ['fooAction', 'barAction']]);  
  
        $this->middleware('subscribed', ['except' => ['fooAction', 'barAction']]);  
    }  
  
}
```

8.2 Validación de formularios

- Laravel incluye una forma sencilla de validar datos y mostrar mensajes usando la clase **Validation**.
 - Cuando recibimos los datos podemos establecer reglas de validación usando dicha clase.
 - Documentación Oficial sobre Validation: <https://laravel.com/docs/5.4/validation>
 - Reglas de validación disponibles: <https://laravel.com/docs/5.4/validation#available-validation-rules>
-

- **Ejemplo de validación extraído y adaptado de:** <https://scotch.io/tutorials/laravel-form-validation>
- Nos aseguramos que en el fichero `.env` tenemos bien configurada la base de datos.

- Creamos primero el **modelo y migración** para la tabla de este ejemplo:

```
php artisan make:model Duck

#Model created successfully.
#Created Migration: 2015_04_24_184031_create_ducks_table
```

- **Editamos el Modelo app/Duck.php:**

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Duck extends Model
{
    protected $table = 'ducks';
    protected $fillable = ['name', 'email', 'password'];
}
```

- Creamos la migración.

```
php artisan make:migration create_ducks_table

# Created Migration: 2017_04_25_110224_create_ducks_table
```

- **Editamos la migración database/migrations/2017_04_25_110224_create_ducks_table:**

```
<?php

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

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

    /**
     * Reverse the migrations.
     *
     */
}
```

```

    * @return void
    */
    public function down()
    {
        Schema::drop('ducks');
    }
}

```

- **Ejecutamos la migración:**

```

php artisan migrate
# Migrating: 2017_04_25_110224_create_ducks_table
# Migrated: 2017_04_25_110224_create_ducks_table

```

- **Código fuente de la Vista `app/views/duck-form.blade.php`:**

```

<!doctype html>
<html>
<head>
    <title>Laravel Form Validation!</title>

    <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
    <style>
        body { padding-bottom:40px; padding-top:40px; }
    </style>
</head>
<body class="container">
    <div class="row">
        <div class="col-sm-8 col-sm-offset-2">

            <div class="page-header">
                <h1><span class="glyphicon glyphicon-flash"></span> Ducks Fly!</h1>
            </div>

            <form method="POST" action="/ducks" novalidate>

                <div class="form-group">
                    <label for="name">Name</label>
                    <input type="text" id="name" class="form-control" name="name" placeholder="Somebody Important" required/>
                </div>

                <div class="form-group">
                    <label for="email">Email</label>
                    <input type="email" id="email" class="form-control" name="email" placeholder="super@cool.com" required/>
                </div>

                <div class="form-group">
                    <label for="password">Password</label>
                    <input type="password" id="password" class="form-control" name="password" required/>
                </div>

                <div class="form-group">
                    <label for="password_confirm">Confirm Password</label>
                    <input type="password" id="password_confirm" class="form-control" name="password_confirm" required/>
                </div>

                <input type="hidden" name="_token" value="{{ csrf_token() }}" />

                <button type="submit" class="btn btn-success">Go Ducks Go!</button>

            </form>

        </div>
    </body>
</html>

```

- **Código fuente de las rutas que muestran y reciben datos por POST del formulario:**

```

<?php
# Editamos el fichero routes/web.php y añadimos las siguientes rutas:

```

```
Route::get('ducks', 'DucksController@mostrar');
Route::post('ducks', 'DucksController@almacenar');
```

- Creamos el controlador **DucksController**:

```
php artisan make:controller DucksController
```

- Editamos el controlador **DucksController**:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use Illuminate\Support\Facades\Hash;
use App\Duck;

class DucksController extends Controller
{
    public function mostrar()
    {
        return View('duck-form');
    }

    public function almacenar(Request $request)
    {
        // Reglas de validación -----
        $reglas = array(
            'name'          => 'required',
            'email'         => 'required|email|unique:ducks',      // obligatorio y debe ser único en la tabla "ducks"
            'password'      => 'required',
            'password_confirm' => 'required|same:password'        // obligatorio y debe ser igual que el campo "password".
        );

        // Si deseamos especificar el error en otro idioma, creamos un array
        // con los mensajes.
        /*
        $mensajes= array(
            'required'=> 'El campo :attribute es obligatorio.',
            'email' => 'El campo :attribute no es correcto.',
            'unique' => 'El campo :attribute está duplicado en la tabla.',
            'same' => 'Las contraseñas no coinciden'
        );
        */

        // Validamos las reglas.
        // Si falla la validación redirecciona automáticamente al formulario origen enviando los errores.
        // $this->validate($request,$reglas,$mensajes);
        $this->validate($request,$reglas);

        // Creamos el registro del nuevo duck.
        $duck = new Duck;
        $duck->name      = $request->input('name');
        $duck->email     = $request->input('email');
        $duck->password  = Hash::make($request->input('password'));

        // Grabamos los datos en la base de datos.
        $duck->save();

        // Almacenamos una variable de sesión Flash
        $request->session()->flash('estado', 'Datos almacenados correctamente!');

        // Redireccionamos de nuevo al formulario.
        return redirect('ducks');
    }
}
```

- Si queremos mostrar todos los errores de validación de forma genérica en la Vista del Formulario de una forma sencilla, podemos añadir los errores antes del formulario:


```

.....

@if(count($errors))
<div class="alert alert-danger">
  <strong>Problema!</strong> Hay fallos en los campos de entrada.
  <br/>
  <ul>
    @foreach($errors->all() as $error)
      <li>{{ $error }}</li>
    @endforeach
  </ul>
</div>
@endif

<form method="POST" action="/ducks" novalidate>
.....

```

• **Información de cómo acceder a los errores devueltos en la Validación:**

<https://laravel.com/docs/5.4/validation#working-with-error-messages>.

• **Código Final de la Vista con la muestra de errores genérica y específica:**

```

<!doctype html>
<html>
<head>
  <title>Laravel Form Validation!</title>

  <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">
  <style>
    body { padding-bottom:40px; padding-top:40px; }
  </style>
</head>
<body class="container">
  <div class="row">
    <div class="col-sm-8 col-sm-offset-2">

      <div class="page-header">
        <h1><span class="glyphicon glyphicon-flash"></span> Ducks Fly!</h1>
      </div>

      @if (Session::has('estado'))
      <div class="alert alert-success">
        <strong>Enhorabuena!</strong> {{ Session::get('estado') }}
      </div>
      @endif

      @if(count($errors))
      <div class="alert alert-danger">
        <strong>Problema!</strong> Hay fallos en los campos de entrada.
        <br/>
        <ul>
          @foreach($errors->all() as $error)
            <li>{{ $error }}</li>
          @endforeach
        </ul>
      </div>
      @endif

      <form method="POST" action="/ducks" novalidate>

        <div class="form-group {{ $errors->has('name') ? 'has-error': '' }}">
          <label for="name">Name</label>
          <input type="text" id="name" class="form-control" name="name" placeholder="Somebody Important" value="{{ old('name') }}">
          <span class="text-danger">{{ $errors->first('name') }}</span>
        </div>

        <div class="form-group {{ $errors->has('email') ? 'has-error': '' }}">
          <label for="email">Email</label>
          <input type="email" id="email" class="form-control" name="email" placeholder="super@cool.com" value="{{ old('email') }}">
          <span class="text-danger">{{ $errors->first('email') }}</span>
        </div>

```

```

<div class="form-group {{$errors->has('name') ? 'has-error': '' }}">
  <label for="password">Password</label>
  <input type="password" id="password" class="form-control" name="password" required/>
  <span class="text-danger">{{ $errors->first('password') }}</span>
</div>

<div class="form-group {{$errors->has('name') ? 'has-error': '' }}">
  <label for="password_confirm">Confirm Password</label>
  <input type="password" id="password_confirm" class="form-control" name="password_confirm" required/>
  <span class="text-danger">{{ $errors->first('password_confirm') }}</span>
</div>

{{ csrf_field() }}

<button type="submit" class="btn btn-success">Go Ducks Go!</button>

</form>

</div>
</div>
</body>
</html>

```

8.2.1 Localización de los mensajes de error al Español (recomendado)

- Nos descargaremos de Github los ficheros con la localización al español: <https://github.com/Laraveles/lang-spanish>
- Una vez descargado el **lang-spanish.zip** copiamos la carpeta es a la ruta **resources/lang**
- Configuramos en el fichero **config/app.php** el **locale** a :

```

.....
'locale' => 'es'
.....

```

8.3 Servidor web con php Artisan, modo mantenimiento, errores, etc. en Laravel 5

8.3.1 Prueba de proyectos con servidor web desde Artisan

- Con Artisan podemos arrancar un servidor web en la carpeta en la que tengamos nuestro proyecto Laravel.
- De esta forma nos evitamos el trasladar a la carpeta web principal para pruebas todo el contenido de un proyecto en particular.
- Simplemente tenemos que entrar en la carpeta dónde está el proyecto Laravel que queremos probar desde la línea de comandos.
- Teclear lo siguiente **php artisan serve** :

```

php artisan serve

// Obtendremos un mensaje como:
Laravel development server started on http://localhost:8000/

```

- Ya solamente nos falta abrir el navegador y acceder a la dirección <http://localhost:8000/> para probar ese proyecto en particular.

8.4 Modo de Mantenimiento

- Podemos **poner un proyecto en mantenimiento** con la siguiente instrucción, accediendo a la carpeta Laravel de ese proyecto desde la línea de comandos:

```
php artisan down
```

- Al intentar acceder se mostrará el mensaje por defecto el mensaje **Be right back**.
- Si queremos **modificar la plantilla de mantenimiento en Laravel 5** tendremos que editar el fichero **resources/views/errors/503.blade.php**:

```

<html>
<head>
<link href="//fonts.googleapis.com/css?family=Lato:100" rel="stylesheet" type="text/css">

```

```

<style>
body {
margin: 0;
padding: 0;
width: 100%;
height: 100%;
color: #B0BEC5;
display: table;
font-weight: 100;
font-family: 'Lato';
}

.container {
text-align: center;
display: table-cell;
vertical-align: middle;
}

.content {
text-align: center;
display: inline-block;
}

.title {
font-size: 72px;
margin-bottom: 40px;
}
</style>
</head>
<body>
<div class="container">
<div class="content">
<div class="title">Estamos en Mantenimiento.<br/>Disculpen las molestias.</div>
</div>
</div>
</body>
</html>

```

- Para activar de nuevo el proyecto desde la línea de comandos:

```
php artisan up
```

8.5 Gestionar errores 404

- Información específica sobre **Gestión de Errores y Logs** en: <http://laravel.com/docs/5.0/errors>
- Para mostrar un **error personalizado 404** crearemos una plantilla en `resources/views/errors/404.blade.php`:

```

<html>
<head>
<link href="//fonts.googleapis.com/css?family=Lato:100" rel="stylesheet" type="text/css">

<style>
body {
margin: 0;
padding: 0;
width: 100%;
height: 100%;
color: #B0BEC5;
display: table;
font-weight: 100;
font-family: 'Lato';
}

.container {
text-align: center;
display: table-cell;
vertical-align: middle;
}

.content {

```

```

text-align: center;
display: inline-block;
}

.title {
font-size: 72px;
margin-bottom: 40px;
}
</style>
</head>
<body>
<div class="container">
<div class="content">
<div class="title">Error: Página no encontrada!<br/>Disculpen las molestias.</div>
</div>
</div>
</body>
</html>

```

8.6 Seeding con Faker

- Es necesario **instalar fzaninotto/faker**.
- Editar **composer.json** y añadir en "require" esta línea: "fzaninotto/faker": "1.*"
- Ejecutar **composer update fzaninotto/faker**
- **Lo normal es usar Faker en los Seeders** (ejemplo en https://manuais.iessanclemente.net/index.php/LARAVEL_Framework_-_Tutorial_01_-_Creaci%C3%B3n_de_API_RESTful#Poblaci.C3.B3n_autom.C3.B3n)
- Ejemplo de **uso de Faker para imprimir datos al azar** en una ruta.

```

Route::get('faker', function()
{
// Documentación sobre Faker: https://github.com/fzaninotto/Faker
// Creamos una instancia de Faker que use español para generar textos.
$faker = Faker\Factory::create('es_ES');

// Creamos un bucle para imprimir 5 registros.
for ($i=0; $i<4; $i++)
{
echo $faker->title.' '. $faker->firstName."<br/>";
echo $faker->address.' -- '. $faker->state."<br/>";
echo $faker->email."<br/>";
echo $faker->randomNumber(3)."<br/>";
echo $faker->year."<br/>";
echo $faker->creditCardType." ". $faker->creditCardNumber."<br/>";
echo $faker->sentence(6)."<br/>";

echo "<hr/>";
}

});

// Daría como resultado algo como ésto:

Lic. Celia
Carrer Iria, 44, 3º E, 41951, Los Delgado -- León
marcos.almonte@marreromatias.es
39
2007
Visa 5346341030623077
Odio eos iusto magnam ad voluptatem.
-----
Ing. Marcos
Calle Lucia, 4, 6º A, 20113, Vega Baja -- Cuenca
joseantonio.almaraz@hotmail.es
997
1974
American Express 5142769205114224
Ut voluptas quod natus dicta velit dolor.
-----
Dr. Rafael
Carrer Jorge, 397, 99º E, 31021, Las Gaona -- A Coruña

```

```
florez.marta@villalpandopuente.org
208
2003
American Express 4556069163893
Quia accusamus quia ab aliquid quibusdam ea voluptatibus.
-----
Dn. Claudia
Camiño Ian, 8, 9° A, 97097, Salcido del Vallès -- Valencia
paola.puig@samaniego.com
34
1974
Visa 4539697037331
Id soluta quia deleniti illum quam et voluptas.
-----
```

8.7 Crear Migrations con definición de campos con PHP Artisan

- Información de cómo hacerlo en: <https://github.com/laracasts/Laravel-5-Generators-Extended>
- Ejemplo de creación de una tabla users:

```
php artisan make:migration:schema create_users_table --schema="username:string, email:string:unique"

// Ejemplos de campos:
username:string
body:text
age:integer
published_at:date
excerpt:text:nullable
email:string:unique:default('foo@example.com')
```

8.8 Permitir peticiones AJAX CORS (Cross-Origin Http Request)

Para permitir peticiones AJAX a nuestro dominio desde cualquier origen, tendríamos que enviar una cabecera de este estilo:

```
# En PHP
header("Access-Control-Allow-Origin: *");

# En Laravel por ejemplo:
Response->header('Access-Control-Allow-Origin','*');
```

8.9 Configuración servidor de correo GMAIL en Laravel

```
# Para configurar el
```

8.10 Ampliación de tiempo máximo token CSRF en Laravel

```
# Hay que acceder a config/session.php
# Modificaremos los siguientes parámetros:

'lifetime' => 120,
'expire_on_close' => false,

# Para evitar el error de "Not Found CSRF Token o similar".
# Dentro de app/exceptions/handler.php
....
```

Veiga (discusión) 15:39 25 abr 2017 (CEST)