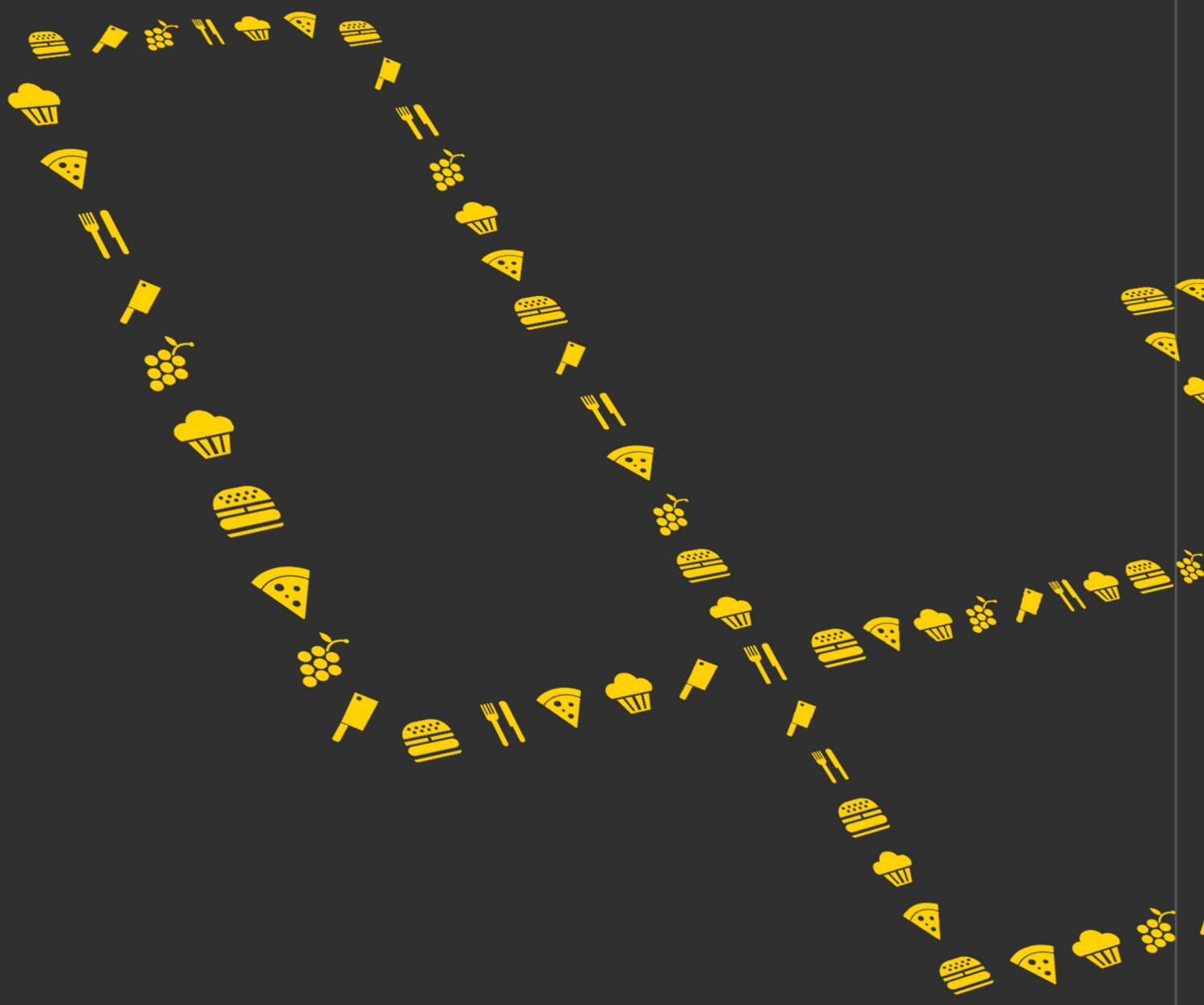

LARAVEL 4
COOKBOOK

BY CHRISTOPHER PITT

Laravel 4 Cookbook (ES)

Christopher Pitt, Taylor Otwell y Carlos Ufano

Este libro está a la venta en http://leanpub.com/laravel4cookbook-es

Esta versión se publicó en 2014-07-04

¡Twitea sobre el libro!

Por favor ayuda a Christopher Pitt, Taylor Otwell y Carlos Ufano hablando sobre el libro en Twitter!

El tweet sugerido para este libro es:

Acabo de comprar el libro Laravel 4 Cookbook (ES)

El hashtag sugerido para este libro es #laravel4cookbook.

Descubre lo que otra gente está diciendo sobre el libro haciendo click en este enlace para buscar el hashtag en Twitter:

https://twitter.com/search?q=#laravel4cookbook

Índice general

Instalando Laravel 4

Laravel 4 utiliza Composer para gestionar sus dependencias. Puedes instalar Composer siguiendo las instrucciones en **http://getcomposer.org/doc/00-intro.md#installation-nix**.

Una vez que tengas Composer funcionando, haz un nuevo directorio o navega hasta uno ya existente e instala Laravel 4 con el siguiente comando:

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

Si has escogido no instalar Composer globalmente (aunque realmente deberías), entonces el comando a utilizar debe ser similar al siguiente:

```
php composer.phar create-project laravel/laravel ./ --prefer-dist
```

Ambos comandos iniciarán el proceso de instalación de Laravel 4. Hay muchas dependencias que tienen que ser seleccionadas y descargadas, por lo que este proceso puede tomar algún tiempo en terminar.

Autenticación

Si eres como yo, habrás malgastado mucho tiempo construyendo sistemas protegidos por contraseña. Solía temer el punto en el que tenía que atornillar el sistema de autenticación a un CMS (Sistema de Gestión de Contenidos) o carrito de compras. Esto era hasta que aprendí lo fácil que era con Laravel 4.

> El código de este capítulo puede encontrarse en: **https://github.com/formativ/tutorial-laravel-4-authentication**

Configurando la Base de datos

Una de las mejores maneras de gestionar usuarios y autenticación es almacenándolos en una base de datos. Los mecanismos de autenticación por defecto de Laravel 4 asumen que usarás alguna forma de almacenamiento en base de datos, y proporciona dos controladores con los que esos usuarios de la base de datos pueden ser recuperados y autenticados.

Conexión a la base de datos

Para usar cualquiera de los controladores proporcionados, primero necesitamos una conexión válida con la base de datos. Ponla en marcha configurando las secciones en el fichero **app/config/database.php**. Aquí hay un ejemplo de la base de datos MySQL que uso para pruebas:

```
1  <?php
2
3  return [
4      "fetch"       => PDO::FETCH_CLASS,
5      "default"     => "mysql",
6      "connections" => [
7          "mysql" => [
8                  "driver"    => "mysql",
9                  "host"      => "localhost",
10                 "database"  => "tutorial",
11                 "username"  => "dev",
12                 "password"  => "dev",
```

```
13                 "charset"   => "utf8",
14                 "collation" => "utf8_unicode_ci",
15                 "prefix"    => ""
16             ]
17     ],
18     "migrations" => "migration"
19 ];
```

Este fichero debería ser guardado como **app/config/database.php**.

He quitado los comentarios, líneas extrañas y opciones de configuración del controlador superfluas.

Controlador de base de datos

El primer controlador que proporciona Laravel 4 se llama **database**. Como su nombre sugiere, este controlador consulta la base de datos directamente a fin de determinar si existen usuarios que coincidan con las credenciales proporcionadas, y si se han proporcionado las credenciales de autenticación apropiadas.

Si este es el controlador que quieres usar, necesitarás la siguiente tabla en la base de datos que tengas configurada:

```
1 CREATE TABLE `user` (
2     `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
3     `username` varchar(255) DEFAULT NULL,
4     `password` varchar(255) DEFAULT NULL,
5     `email` varchar(255) DEFAULT NULL,
6     `created_at` datetime DEFAULT NULL,
7     `updated_at` datetime DEFAULT NULL,
8     PRIMARY KEY (`id`)
9 ) CHARSET=utf8 COLLATE=utf8_unicode_ci;
```

> Aquí, y más adelante, me desvío del estándar de los nombres plurales de tablas de base de datos. Normalmente, recomendaría quedarse con el estándar, pero esto me dió la oportunidad de demostrar como puedes configurar nombres de tablas de base de datos, tanto en migraciones como en modelos.

Controlador Eloquent

El segundo controlador que Laravel 4 proporciona se llama **eloquent**. Eloquent es el nombre del ORM (mapeo objeto-relacional) que también Laravel 4 proporciona, para abstraer datos del modelo. Es similar en que consultará una base de datos para determinar si un usuario es auténtico, pero el interfaz que utiliza para hacer esa determinación es un poco diferente a las consultas directas a base de datos.

Si estás construyendo aplicaciones medianas a grandes, usando Laravel 4, tienes una buena oportunidad para usar modelos Eloquent para representar objetos de base de datos. Con esto en mente, voy a dedicar algún tiempo a a la elaboración de la participación de modelos Eloquent en el proceso de autenticación.

> Si quieres ignorar todas estas cosas de Eloquent, siéntete libre de saltar las siguientes secciones que se ocupan de las migraciones y modelos.

Creando una migración

Puesto que estamos usando Eloquent para gestionar cómo nuestra aplicación se comunica con la base de datos; podemos también usar las herramientas de manipulación de tablas de base de datos de Laravel 4.

Para empezar, ve a la raíz de tu proyecto y escribe el siguiente comando:

```
1 php artisan migrate:make --table="user" CreateUserTable
```

> El argumento **–table=”user”** coincide con la propiedad **$table=user** que definiremos en el modelo **User**.

Esto generará el andamiaje para la tabla de usuarios, que debería parecerse a lo siguiente:

```php
<?php

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

class CreateUserTable
extends Migration
{
    public function up()
    {
        Schema::table('user', function(Blueprint $table)
        {
            //
        });
    }
    public function down()
    {
        Schema::table('user', function(Blueprint $table)
        {
            //
        });
    }
}
```

Este fichero debería guardarse como **app/database/migrations/0000_00_00_000000_CreateUserTable.php**. El tuyo puede ser un poco diferente, donde los 0 se sustituyen por otros números.

El esquema de nombrado de ficheros puede parecer extraño, pero es por una buena razón. Los sistemas de migración están diseñados para ejecutarse en cualquier servidor, y el orden en que se deben ejecutar es fijo. Todo esto para permitir cambios en la base de datos para estar bajo control de versiones.

La migración se crea solo con el andamiaje más básico, que significa que necesitamos añadir los campos en la tabla de usuarios:

```php
<?php

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

class CreateUserTable
extends Migration
{
    public function up()
    {
        Schema::create("user", function(Blueprint $table)
        {
            $table->increments("id");

            $table
                ->string("username")
                ->nullable()
                ->default(null);

            $table
                ->string("password")
                ->nullable()
                ->default(null);

            $table
                ->string("email")
                ->nullable()
                ->default(null);

            $table
                ->dateTime("created_at")
                ->nullable()
                ->default(null);

            $table
                ->dateTime("updated_at")
                ->nullable()
                ->default(null);
        });
    }

    public function down()
```

```
43      {
44          Schema::dropIfExists("user");
45      }
46  }
```

> Este fichero debería guardarse como **app/database/migrations/0000_00_00_000000_CreateUserTable.php**. El tuyo puede ser un poco diferente, donde los 0 se sustituyen por otros números.

Aquí, hemos añadido campos para id, nombre de usuario, contraseña, fecha de creación y fecha de actualización. Hay métodos para acortar los campos de tiempo, pero prefiero añadir estos campos explícitamente. Todos los campos pueden ser nulos y su valor por defecto es null.

También hemos añadido el método de borrado, que se ejecutará si las migraciones se invierten, y eliminará la tabla de usuarios si existe.

> Las formas cortas para añadir los campos de tiempo pueden encontrarse en: **http://laravel.com/docs/schema#adding-columns**

Esta migración funcionará, incluso si solo quieres usar el controlador de base de datos, pero es por lo general parte de una instalación más grande, que incluye modelos y sembradoras (seeders).

Creando un modelo

Laravel 4 proporciona un modelo **User**, con todos los métodos de interfaz que requiere. Lo he modificado ligeramente, pero los fundamentos siguen ahí...

```
1   <?php
2
3   use Illuminate\Auth\UserInterface;
4   use Illuminate\Auth\Reminders\RemindableInterface;
5
6   class User
7   extends Eloquent
8   implements UserInterface, RemindableInterface
9   {
10      protected $table  = "user";
11      protected $hidden = ["password"];
12
```

```
13      public function getAuthIdentifier()
14      {
15          return $this->getKey();
16      }
17
18      public function getAuthPassword()
19      {
20          return $this->password;
21      }
22
23      public function getReminderEmail()
24      {
25          return $this->email;
26      }
27  }
```

Este fichero debería guardarse como **app/models/User.php**.

Observa la propiedad **$table=user** que hemos definido. Debe coincidir con la tabla que definimos en nuestras migraciones.

El modelo **User** extiende **Eloquent** e implementa dos interfaces que aseguran que el modelo es válido para operaciones de autenticación y recordatorio. Nos ocuparemos de los interfaces más tarde, pero es importante notar los métodos que esas interfaces requieren.

Laravel 4 permite el uso de cualquier dirección de email o nombre de usuario para identificar al usuario, pero es un campo diferente de lo que devuelve **getAuthIdentifier()**. El interfaz **UserInterface** especifica el nombre de campo contraseña, pero esto puede ser modificado sobreescribiendo/cambiando el método **getAuthPassword()**.

El método **getReminderEmail()** devuelve una dirección de email en la que contactar al usuario con un email de reinicio de contraseña, si esto fuese necesario.

Eres libre para especificar cualquier personalización del modelo, sin temor a que se rompan los mecanismos de autenticación integrados.

Creando una sembradora (seeder)

Laravel 4 también incluye un sistema de siembra, que puede utilizarse para añadir registros a tu base de datos después de la migración inicial. Para añadir los usuarios iniciales a mi proyecto, tengo la siguiente clase de sembradora:

```
<?php

class UserSeeder
extends DatabaseSeeder
{
    public function run()
    {
        $users = [
            [
                "username" => "christopher.pitt",
                "password" => Hash::make("7h3 iMOST!53cu23"),
                "email"    => "chris@example.com"
            ]
        ];

        foreach ($users as $user)
        {
            User::create($user);
        }
    }
}
```

Este fichero debería guardarse como **app/database/seeds/UserSeeder.php**.

Ejecutando esto añadirá mi cuenta de usuario a la base de datos, pero para poder ejecutarlo necesitamos añadirlo a la principal clase **DatabaseSeeder**:

```php
<?php

class DatabaseSeeder
extends Seeder
{
    public function run()
    {
        Eloquent::unguard();
        $this->call("UserSeeder");
    }
}
```

Este campo debería guardarse como **app/database/seeds/DatabaseSeeder.php**.

Ahora, cuando la clase **DatabaseSeeder** sea invocada, sembrará la tabla de usuarios con mi cuenta. Si ya has configurado tu migración y modelo, y proporcionado datos de la conexión a la base de datos, los siguientes comandos deberían entonces poner todo en marcha y funcionando.

```
composer dump-autoload
php artisan migrate
php artisan db:seed
```

El primer comando asegura que todas las nuevas clases que hemos creado sean correctamente cargadas de manera automática. El segundo cra las tablas de la base de datos específicas para la migración. El tercero siembra los datos del usuario en la tabla de usuarios.

Configurando la autenticación

Las opciones de configuración de los mecanismos de autenticación son escasas, pero sí permiten cierta personalización.

```php
<?php

return [
    "driver"   => "eloquent",
    "model"    => "User",
    "reminder" => [
        "email"  => "email.request",
        "table"  => "token",
        "expire" => 60
    ]
];
```

Este fichero debe ser guardado como **app/config/auth.php**.

Todos estos valores son importnates, y la mayoría auto-explicatorios y fáciles de entender. La vista utilizada para componer la solicitud del email se especifica con **email** ⇒ **email.request** y el tiempo en el que el token de reinicio caducará se especifica con **expire** ⇒ **60**.

Presta especial atención a la vista especificada por **email** ⇒ **email.request**—le dice a Laravel que cargue el fichero **app/views/email/request.blade.php** en vez del **app/views/emails/auth/reminder.blade.php** por defecto.

Hay varias cosas que se beneficiarían de las opciones de configuración, que actualmente están siendo programadas en el código de los proveedores. Veremos algunas de ellas, a medida que vayan surgiendo.

Iniciando sesión

Para permitir autenticarse a los usuarios para usar nuestra aplicación, vamos a construir una página de acceso, donde los usuarios puedan introducir sus datos de inicio de sesión. Si sus datos son válidos, serán redirigidos a su página de perfil.

Creación de una vista de diseño

Antes de crear cualquiera de las páginas de nuestra aplicación, sería consejable abstraer todo nuestro marcado de diseño y estilo. Para ello, vamos a crear una vista de diseño con varios includes, usando el motor de plantillas Blade.

En primer lugar, tenemos que crear la vista de diseño.

```
<!DOCTYPE html>
<html lang=”en”>
    <head>
        <meta charset="UTF-8" />
        <link
            type="text/css"
            rel="stylesheet"
            href="/css/layout.css" />
        <title>
            Tutorial
        </title>
    </head>
    <body>
        @include("header")
        <div class="content">
            <div class="container">
                @yield("content")
            </div>
        </div>
        @include("footer")
    </body>
</html>
```

Este fichero debería guardarse como **app/views/layout.blade.php**.

La vista de diseño es principalmente HTML estándar, con dos etiquetas específicas de Blade en ella. Las etiquetas **@include()** le dicen a Laravel que incluya las vistas (nombradas en estas cadenas como **header** y **footer**) del directorio de vistas.

¿Has notado que hemos omitido la extensión **.blade.php**? Laravel la añade automáticamente por nosotros. También une los datos proporcionados por ambos includes a la vista de diseño.

La segunda etiqueta Blade es **yield()**. Esta etiqueta acepta un nombre de sección, y muestra los datos almacenados en esa sección. Las vistas en nuestra aplicación extenderán esta vista de diseño,

mientras especifican sus propias secciones **content** para que su marcado sea embebido en el marcaod del diseño. Verás como se definen exactamente las secciones en breve.

```
@section("header")
    <div class="header">
        <div class="container">
            <h1>Tutorial</h1>
        </div>
    </div>
@show
```

Este fichero debería guardarse como **app/views/header.blade.php**.

El fichero header del include contiene dos etiquetas Blade que, en conjunto, indican a Blade que almacene el marcado en la sección que lo nombre, y lo renderice en la plantilla.

```
@section("footer")
    <div class="footer">
        <div class="container">
            Powered by <a href="http://laravel.com/">Laravel</a>
        </div>
    </div>
@show
```

Este fichero debería guardarse como **app/views/footer.blade.php**.

Del mismo modo, el include footer envuelve su marcado en la sección que lo nombre e inmediatamente lo renderiza en la plantilla.

Puede que te estés preguntando porqué necesitamos envolver el marcado, de estos ficheros include, en secciones. Estamos renderizádolos de manera inmediata, después de todo. Haciéndolo así permitimos poder alterar su contenido. Lo veremos pronto en acción.

```
body
{
    margin      : 0;
    padding     : 0 0 50px 0;
    font-family : "Helvetica", "Arial";
    font-size   : 14px;
    line-height : 18px;
    cursor      : default;
}
a
{
    color : #ef7c61;
}
.container
{
    width    : 960px;
    position : relative;
    margin   : 0 auto;
}
.header, .footer
{
    background  : #000;
    line-height : 50px;
    height      : 50px;
    width       : 100%;
    color       : #fff;
}
.header h1, .header a
{
    display : inline-block;
}
.header h1
{
    margin      : 0;
    font-weight : normal;
}
.footer
{
    position : absolute;
    bottom   : 0;
}
.content
```

```
{
    padding : 25px 0;
}
label, input, .error
{
    clear   : both;
    float   : left;
    margin : 5px 0;
}
.error
{
    color : #ef7c61;
}
```

Este fichero debería guardarse como **public/css/layout.css**.

Terminamos añadiendo algunos estilos básicos, que vincularemos en el elemento **head**. Estos alteran las fuentes por defecto y el diseño. Tu aplicación debería funcionar aún sin ellos, pero se mostraría todo un poco desordenado.

Creando una vista de acceso

La vista de acceso es esencialmente un formulario, en el que los usuarios introducen sus credenciales.

```
@extends("layout")
@section("content")
    {{ Form::open([
        "route"        => "user/login",
        "autocomplete" => "off"
    ]) }}
        {{ Form::label("username", "Username") }}
        {{ Form::text("username", Input::old("username"), [
            "placeholder" => "john.smith"
        ]) }}
        {{ Form::label("password", "Password") }}
        {{ Form::password("password", [
            "placeholder" => "□□□□□□□□□□□□"
        ]) }}
        {{ Form::submit("login") }}
```

```
16      {{ Form::close() }}
17  @stop
18  @section("footer")
19      @parent
20      <script src="//polyfill.io"></script>
21  @stop
```

Este fichero debería guardarse como **app/views/user/login.blade.php**.

La primera etiqueta Blade, en la vista de acceso, le indica aLaravel que esta vista extiende la vista de diseño. La segunda le dice qué marcado incluir en la sección de contenido. Estas etiquetas formarán la base de todas las vistas que creemos (además de las de diseño).

Luego utilizamos **{{** y **}}** para decirle a Laravel que queremos que el código contenido sea interpretado como PHP. Abrimos el formulario con el método **Form::open()**, proporcionando una ruta para que el formulario envíe su contenido mediante post, además de parámetros opcionales como segundo argumento.

Entonces definimos dos labels (etiquetas de campo) y tres inputs (campos) como parte del formulario. Las labels aceptan un argumento nombre, seguido por un argumento de texto. El siguiente input acepta un argumento nombre, un argumento valor y parámetros opcionales. El input password (contraseña) acepta un argumento nombre y parámetros opcionales. Por último, el input submit (entregar) acepta un argumento nombre y un argumento texto (como las labels).

Cerramos el formulario con una llamada a **Form::close()**.

Puedes encontrar más información sobre los métodos **Form** que Laravel ofrece en: **http://laravel.com/docs/html**

La última parte de la vista de acceso es donde sobreescribimos el marcado del pie por defecto (especificado en el include footer que creamos anteriormente). Usamos el mismo nombre de sección, pero no terminamos la sección con **@show**. Ya se renderizará debido a cómo hemos definido el include, por lo que solo usamos **@stop** de la misma forma como cerramos la sección de contenido.

También utilizamos la etiqueta Blade **@parent** para decirle a Laravel que queremos que se muestre el marcado que definimos en el pie por defecto. No estamos cambiándolo completamente, simplemente añadiendo una etiqueta script.

> Puede sencontrar más información sobre las etiquetas Blade en: **http://laravel.com/docs/templates#blade-templating**

El script que hemos incluido se llama polyfill.io. Es una colección de cuñas de navegador permitiendo cosas como el atributo **placeholder** (que no están siempre presente en viejos navegadores).

> Puedes encontrar más información sobre Polyfill.io en: **https://github.com/jonathantneal/polyfill**

Nuestra vista de acceso está ahora completada, pero básicamente es inútil sin el código en la parte del servidor que acepte la entrada de datos y devuelva un resultado. Vamos a resolverlo!

Creando una acción de acceso

La acción de acceso es lo que pega la lógica de autenticación con las vistas que hemos creado. Si has estado siguiéndonos desde el principio, es posible que te hayas preguntado si íbamos a probar cualquiera de estas cosas en un navegador. Hasta este punto, no había forma de decirle a nuestra aplicación que cargase esta vista.

Para empezar, tenemos que agregar una ruta para la acción de acceso.

```
1  <?php
2
3  Route::any("/", [
4      "as"   => "user/login",
5      "uses" => "UserController@loginAction"
6  ]);
```

> Este fichero debería guardarse como **app/routes.php**.

El fichero de rutas muestra una página de apoyo para una nueva aplicación Laravel 4, renderizando una vista directamente. Necesitamos cambiar eso para usar un controlador/acción. No es que tengamos, podríamos realizar fácilmente la lógica en el archivo de rutas, pero simplemente no sería muy ordenado.

Especificamos un nombre para la ruta con **as** ⇒ **user/login**, y le damos un destino con **uses** ⇒ **UserController@loginAction**. Este coincidirá con todas las llamadas a la ruta por defecto /, e incluso tiene un nombre que puede utilizarse para hacer referencia a esta ruta con facilidad.

Lo siguiente, necesitamos crear el controlador.

```
<?php

class UserController
extends Controller
{
    public function loginAction()
    {
        return View::make("user/login");
    }
}
```

Este fichero debería guardarse como **app/controllers/UserController.php**.

Definimos el **UserController** (para extender la clase **Controller**). En él, tenemos el único método **loginAction()** que especificamos en el fichero de rutas. Todo esto actualmente hace renderizar la vista de acceso en el navegador, ¡pero es suficiente para que seamos capaces de ver nuestro progreso!

Autenticando a usuarios

Bien, como tenemos el formulario, ahora necesitamos conectarlo a la base de datos para que podamos autenticar correctamente a los usuarios.

```
<?php

class UserController
extends Controller
{
    public function loginAction()
    {
        if (Input::server("REQUEST_METHOD") == "POST")
        {
            $validator = Validator::make(Input::all(), [
                "username" => "required",
```

```
12                  "password" => "required"
13              ]);
14
15              if ($validator->passes())
16              {
17                  echo "Validation passed!";
18              }
19              else
20              {
21                  echo "Validation failed!";
22              }
23          }
24
25          return View::make("user/login");
26      }
27  }
```

Este fichero debería guardarse como **app/controllers/UserController.php**.

Nuestra clase **UserController** ha cambiado algo. En primer lugar, tenemos que actuar sobre los datos que se envían al método **loginAction()** vía post; y para hacer esto comprobamos la propiedad **REQUEST_METHOD** del servidor. Si su valor es **POST** podemos asumir que el formulario ha sido enviado vía post a esta acción, y procederemos entonces con la fase de validación.

También es común ver las acciones POST y GET de forma independiente para la misma página. Si bien esto hace las cosas más ordenadas, y evita la necesidad de comprobar la propiedad **REQUEST_METHOD**, yo prefiero manejar ambas en la misma acción.

Laravel 4 ofrece un gran sistema de validación, y una de las maneras de usarlo es llamando al método **Validator::make()**. El primer argumento es una matriz de datos a validar, y el segundo argumento es una matriz de reglas.

Solo hemos especificado que los campos username y password son obligatorios, pero hay otras muchas reglas de validación (algunas de las cuales usaremos en un rato). La clase **Validator** también tiene un método **passes()**, que usamos para conocer si los datos del formulario enviados son válidos.

> A veces es mejor almacenar la lógica de validación fuera del controlador. A menudo la pongo en el modelo, pero puedes también crear una clase específica para manipulación y validación de la entrada.

Si envías este formulario, ahora te dirá si los campos obligatorios se han introducido o no, pero hay una forma más elegante de mostrar esta clase de mensajes...

```php
<?php

use Illuminate\Support\MessageBag;

class UserController
extends Controller
{
    public function loginAction()
    {
        $data = [];

        if (Input::server("REQUEST_METHOD") == "POST")
        {
            $validator = Validator::make(Input::all(), [
                "username" => "required",
                "password" => "required"
            ]);

            if ($validator->passes())
            {
                //
            }
            else
            {
                $data["errors"] = new MessageBag([
                    "password" => [
                        "Username and/or password invalid."
                    ]
                ]);
            }
        }

        return View::make("user/login", $data);
```

```
34      }
35  }
```

Este fichero debería guardarse como **app/controllers/UserController.php**.

Con los cambios de anteriores, estamos utilizando la clase **MessageBag** para almacenar mensajes de error de validación. Esto es similar a como guarda implícitamente la clase Validation sus errores, pero en vez de mostrar mensajes de error individuales para cada nombre de usuario o contraseña, estamos mostrando un único mensaje de error para ambos. ¡Los formularios de acceso son un poco más seguros de esta forma!

Para mostrar este mensaje de error, necesitaremos cambiar la vista de acceso.

```
1  @extends("layout")
2  @section("content")
3      {{ Form::open([
4          "route"        => "user/login",
5          "autocomplete" => "off"
6      ]) }}
7          {{ Form::label("username", "Username") }}
8          {{ Form::text("username", Input::get("username"), [
9              "placeholder" => "john.smith"
10         ]) }}
11         {{ Form::label("password", "Password") }}
12         {{ Form::password("password", [
13             "placeholder" => "••••••••••"
14         ]) }}
15         @if ($error = $errors->first("password"))
16             <div class="error">
17                 {{ $error }}
18             </div>
19         @endif
20         {{ Form::submit("login") }}
21     {{ Form::close() }}
22 @stop
23 @section("footer")
24     @parent
25     <script src="//polyfill.io"></script>
26 @stop
```

Este fichero debería guardarse como **app/views/user/login.blade.php**.

Como probablemente puedes ver, hemos añadido una comprobación de la existencia del mensaje de error, y lo hemos renderizado con un elemento div con estilo. Si la validación falla, ahora verás el mensaje de error debajo del campo de contraseña.

Redirigiendo con input

Uno de los errores comunes de los formularios es que a menudo refrescan la página si reenvían el formulario. Podemos superar esto con un poco de la magia de Laravel. ¡Almacenaremos los datos del formulario enviados por post en la sesión, y redirigiremos a la página de acceso!

```
1  <?php
2
3  use Illuminate\Support\MessageBag;
4
5  class UserController
6  extends Controller
7  {
8      public function loginAction()
9      {
10         $errors = new MessageBag();
11
12         if ($old = Input::old("errors"))
13         {
14             $errors = $old;
15         }
16
17         $data = [
18             "errors" => $errors
19         ];
20
21         if (Input::server("REQUEST_METHOD") == "POST")
22         {
23             $validator = Validator::make(Input::all(), [
24                 "username" => "required",
25                 "password" => "required"
26             ]);
27
```

```
28              if ($validator->passes())
29              {
30                  //
31              }
32              else
33              {
34                  $data["errors"] = new MessageBag([
35                      "password" => [
36                          "Username and/or password invalid."
37                      ]
38                  ]);
39
40                  $data["username"] = Input::get("username");
41
42                  return Redirect::route("user/login")
43                      ->withInput($data);
44              }
45          }
46
47          return View::make("user/login", $data);
48      }
49  }
```

Este fichero debería guardarse como **app/controllers/UserController.php**.

Lo primero que hemos hecho es declarar una nueva instancia de **MessageBag**. Lo hacemos porque la vista todavía comprobará los errores **MessageBag**, por si han sido o no almacenados en la sesión. Si es así, en cambio, en la sesión, sobreescribiremos la nueva instancia que creamos con la instancia almacenada.

Entonces lo añadiremos a la matriz **$data** de modo que se pasa a la vista, y puede ser renderizado.

Si la validación falla, almacenaremos el nombre de usuario en la matriz **$data**, junto con los errores de validación, y redirigiremos de vuelta a la misma ruta (utilizando también el método **withInput()** para almacenar nuestros datos en la sesión).

Nuestra vista permanece sin cambios, pero podemos refrescarla sin el horrible reenvío del formulario(y los molestos mensajes del navegador que van con él).

Autenticando credenciales

El último paso en la autenticación es comprobar los datos proporcionados en el formulario contra la base de datos. Laravel maneja esto fácilmente por nosotros.

```
<?php

use Illuminate\Support\MessageBag;

class UserController
extends Controller
{
    public function loginAction()
    {
        $errors = new MessageBag();

        if ($old = Input::old("errors"))
        {
            $errors = $old;
        }

        $data = [
            "errors" => $errors
        ];

        if (Input::server("REQUEST_METHOD") == "POST")
        {
            $validator = Validator::make(Input::all(), [
                "username" => "required",
                "password" => "required"
            ]);

            if ($validator->passes())
            {
                $credentials = [
                    "username" => Input::get("username"),
                    "password" => Input::get("password")
                ];

                if (Auth::attempt($credentials))
                {
                    return Redirect::route("user/profile");
                }
```

```
39              }
40
41              $data["errors"] = new MessageBag([
42                  "password" => [
43                      "Username and/or password invalid."
44                  ]
45              ]);
46
47              $data["username"] = Input::get("username");
48
49              return Redirect::route("user/login")
50                  ->withInput($data);
51          }
52
53          return View::make("user/login", $data);
54      }
55  }
```

Este fichero debería guardarse como **app/controllers/UserController.php**.

Simplemente necesitamos pasar los datos del formulario enviados por post (**$credentials**) al método **Auth::attempt()** y, si las credenciales son válidas, el usuario accederá iniciando sesión. Si es válido, devolveremos una redirección a la página del perfil del usuario.

También hemos eliminado los códigos de error fuera de la cláusula else. Es así porque ocurrirá en ambos errores de validación y también de autenticación. El mismo mensaje de error (en el caso de páginas de acceso) está muy bien.

Restableciendo contraseñas

¡El mecanismo de restablecimeinto de contraseñas incorporado en Laravel 4 es genial! Vamos a configurarlo para que los usuarios puedan restablecer sus contraseñas simplemente proporcionando su dirección de correo electrónico.

Creando una vista de restablecimiento de contraseña

Necesitamos dos vistas para que los usuarios puedan restablecer sus contraseñas. Necesitamos una vista para que ellos introduzcan su dirección de email y se les envíe un token de restablecimiento, y necesitaremos otra vista para que introduzcan una nueva contraseña para su cuenta.

```
@extends("layout")
@section("content")
    {{ Form::open([
        "route"        => "user/request",
        "autocomplete" => "off"
    ]) }}
        {{ Form::label("email", "Email") }}
        {{ Form::text("email", Input::get("email"), [
            "placeholder" => "john@example.com"
        ]) }}
        {{ Form::submit("reset") }}
    {{ Form::close() }}
@stop
@section("footer")
    @parent
    <script src="//polyfill.io"></script>
@stop
```

Este fichero debería guardarse como **app/views/user/request.blade.php**.

Esta vista es similar a la vista de acceso, exceptuando que tiene un único campo para una dirección de correo electrónico.

```
@extends("layout")
@section("content")
    {{ Form::open([
        "url"          => URL::route("user/reset") . $token,
        "autocomplete" => "off"
    ]) }}
        @if ($error = $errors->first("token"))
            <div class="error">
                {{ $error }}
            </div>
        @endif
        {{ Form::label("email", "Email") }}
        {{ Form::text("email", Input::get("email"), [
            "placeholder" => "john@example.com"
        ]) }}
        @if ($error = $errors->first("email"))
```

```
17              <div class="error">
18                  {{ $error }}
19              </div>
20          @endif
21          {{ Form::label("password", "Password") }}
22          {{ Form::password("password", [
23              "placeholder" => "••••••••••"
24          ]) }}
25          @if ($error = $errors->first("password"))
26              <div class="error">
27                  {{ $error }}
28              </div>
29          @endif
30          {{ Form::label("password_confirmation", "Confirm") }}
31          {{ Form::password("password_confirmation", [
32              "placeholder" => "••••••••••"
33          ]) }}
34          @if ($error = $errors->first("password_confirmation"))
35              <div class="error">
36                  {{ $error }}
37              </div>
38          @endif
39          {{ Form::submit("reset") }}
40      {{ Form::close() }}
41  @stop
42  @section("footer")
43      @parent
44      <script src="//polyfill.io"></script>
45  @stop
```

Este fichero debería guardarse como **app/views/user/reset.blade.php**.

Ok, ahora lo entenderás. Hay un formulario con algunos inputs y mensajes de error. Una cosa importante a tener en cuenta es el cambio en la acción del formulario, a saber, el uso de **URL::route()** en combinación con una variable asignada a la vista. Pondremos eso en la acción, así que no te preocupes por ahora.

He modificado también ligeramente el correo electrónico de solicitud del token de contraseña, aunque sigue siendo casi idéntico al de la vista predeterminada que proporciona una nueva instalación de Laravel 4.

```
<!DOCTYPE html>
<html lang="en">
    <head>
        <meta charset="utf-8" />
    </head>
    <body>
        <h1>Password Reset</h1>
        To reset your password, complete this form:
        {{ URL::route("user/reset") . "?token=" . $token }}
    </body>
</html>
```

Este fichero debería guardarse como **app/views/email/request.blade.php**.

Recuerda que cambiamos las opciones de configuración de envío del correo electrónico de esta vista por el predeterminado en **app/views/emails/auth/reminder.blade.php**.

Creando una acción de restablecimiento de contraseña

A fin de que las acciones sean accesibles, necesitamos añadir rutas para ellas.

```
<?php

Route::any("/", [
    "as"   => "user/login",
    "uses" => "UserController@loginAction"
]);

Route::any("/request", [
    "as"   => "user/request",
    "uses" => "UserController@requestAction"
]);

Route::any("/reset", [
    "as"   => "user/reset",
    "uses" => "UserController@resetAction"
]);
```

Este fichero debería guardarse como **app/routes.php**.

Recuerda; la ruta de petición es para solicitar un token de restablecimiento, y la ruta de restablecimeinto es para restablecer una contraseña.

También necesitamos generar la tabla de tokens de restablecimiento de contraseñas, utilizando artisan.

```
php artisan auth:reminders
```

Esto generará una plantilla de migración para la tabla de recordatorio.

```
<?php

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

class CreateTokenTable
extends Migration
{
    public function up()
    {
        Schema::create("token", function(Blueprint $table)
        {
            $table
                ->string("email")
                ->nullable()
                ->default(null);

            $table
                ->string("token")
                ->nullable()
                ->default(null);

            $table
                ->timestamp("created_at")
                ->nullable()
                ->default(null);
        });
    }
```

```
29
30      public function down()
31      {
32          Schema::dropIfExists("token");
33      }
34  }
```

> Este fichero debería guardarse como **app/database/migrations/0000_00_00_000000_CreateTokenTable.php**. El tuyo podría ser ligeramente diferente cambiando los 0 por otros números.

He modificado ligeramente la plantilla, pero los fundamentos son los mismos. Esto creará una tabla con los campos **email**, **token** y **created_at**, que los mecanismos de autenticación usarán para generar y validar los tokens de restablecimiento de las contraseñas.

Con esto en mente, podemos empezar a añadir nuestras acciones de restablecimiento de contraseña.

```
1   public function requestAction()
2   {
3       $data = [
4           "requested" => Input::old("requested")
5       ];
6
7       if (Input::server("REQUEST_METHOD") == "POST")
8       {
9           $validator = Validator::make(Input::all(), [
10              "email" => "required"
11          ]);
12
13          if ($validator->passes())
14          {
15              $credentials = [
16                  "email" => Input::get("email")
17              ];
18
19              Password::remind($credentials,
20                  function($message, $user)
21                  {
22                      $message->from("chris@example.com");
23                  }
24              );
```

```
25
26              $data["requested"] = true;
27
28              return Redirect::route("user/request")
29                  ->withInput($data);
30          }
31      }
32
33      return View::make("user/request", $data);
34  }
```

Esto se ha extraído de **app/controllers/UserController.php**.

El método **requestAction()** valida los datos enviados del formulario de la misma forma que hacía el método **loginAction()**, pero en vez de pasar los datos del formulario a **Auth::attempt()**, este lo pasa a **Password::remind()**. Este método acepta una matriz de credenciales (que normalmente solo incluye una dirección de correo electrónico), y también permite una devolución de llamada en la que puedes personalizar el correo electrónico que se envía.

```
1   public function resetAction()
2   {
3       $token = "?token=" . Input::get("token");
4
5       $errors = new MessageBag();
6
7       if ($old = Input::old("errors"))
8       {
9           $errors = $old;
10      }
11
12      $data = [
13          "token"  => $token,
14          "errors" => $errors
15      ];
16
17      if (Input::server("REQUEST_METHOD") == "POST")
18      {
19          $validator = Validator::make(Input::all(), [
20              "email"                 => "required|email",
```

```
21              "password"              => "required|min:6",
22              "password_confirmation" => "same:password",
23              "token"                 => "exists:token,token"
24          ]);
25
26          if ($validator->passes())
27          {
28              $credentials = [
29                  "email" => Input::get("email")
30              ];
31
32              Password::reset($credentials,
33                  function($user, $password)
34                  {
35                      $user->password = Hash::make($password);
36                      $user->save();
37
38                      Auth::login($user);
39                      return Redirect::route("user/profile");
40                  }
41              );
42          }
43
44          $data["email"] = Input::get("email");
45
46          $data["errors"] = $validator->errors();
47
48          return Redirect::to(URL::route("user/reset") . $token)
49              ->withInput($data);
50      }
51
52      return View::make("user/reset", $data);
53  }
```

Esto se ha extraído de **app/controllers/UserController.php**.

El método **resetAction()** es más de lo mismo. Lo empezamos creando la cadena de consulta del token (que usaremos para redirecciones, manteniendo el token en todos los estados de la página de restablecimiento). Obtendremos los mensajes de error viejos, como hicimos para la página de acceso, y validaremos los datos enviados del formulario.

Si todos los datos son válidos, los pasaremos a **Password::reset()**. El segundo argumento es la lógica utilizada para actualizar el registro del usuario en la base de datos. Estamos actualizando la contraseña, almacenando el registro y entonces iniciando automáticamente la sesión del usuario.

Si todo esto sale a pedir de boca, redirigiremos a la página del perfil. Si no, redirigiremos de vuelta a la página de restablecimiento, pasando a través de los mensajes de error.

> Hay una cosa extraña sobre los mecanismos de autenticación aquí, los nombres de los campos contraseña/token están incluidos en el código y hay una validación también incluida en el código en la función **Password::reset()** que no utiliza la clase **Validation**. Mientras tus nombres de campos sean **password**, **password_confirmation** y **token**, y tu contraseña mayor de 6 caracteres, no notarás esta situación extraña.
>
> Como alternativa, puedes modificar los nombres de campo y la validación aplicada en el fichero **vendor/laravel/framework/src/Illuminate/Auth/Reminders/PasswordBroker.php** o implementar tu propio **ReminderServiceProvider** para reemplazar lo que ofrece Laravel 4. Los detalles de estos dos enfoques están más allá del alcance de este tutorial. Puedes encontrar detalles para crear proveedores de servicio en el excelente libro de Taylor Otwell, en: **https://leanpub.com/laravel**

> Como mencioné antes, se puede establecer la cantidad de tiempo tras el cual el token de restablecimiento caduque, en el fichero **app/config/auth.php**.

> Puedes encontrar más información acerca de los métodos de autenticación en: **http://laravel.com/docs/security#authen**
> **users**

> Puedes encontrar más información sobre los métodos de correo electrónico en: **http://laravel.com/docs/mail**

Trabajando con usuarios autenticados

Bien. Ya tenemos en nuestro haber el restablecimiento de contraseña y el acceso. La parte final de este tutorial es para que podamos utilizar los datos de sesión en nuestra aplicación, y proteger el

acceso no autenticado para securizar partes de nuestra aplicaciónr.

Creando una página de perfil

Para mostrar algunos de los datos de seión del usuario a los que tenemos acceso, vamos a implementar una vista de perfil.

```
@extends("layout")
@section("content")
    <h2>Hello {{ Auth::user()->username }}</h2>
    <p>Welcome to your sparse profile page.</p>
@stop
```

Este fichero debería guardarse como **app/views/user/profile.blade.php**.

Esta página de perfil increíblemente escasa muestra una sola cosa, se pueden obtener datos del modelo de usuario accediendo al objeto devuelto por el método **Auth::user()**. Cualqueir campo que hayas definido en este modelo (o tabla de base de datos) son accesibles de esta manera.

```
public function profileAction()
{
    return View::make("user/profile");
}
```

Esto se ha extraído de **app/controllers/UserController.php**.

El método **profileAction()** es tan simple como la vista. No necesitamos pasar ningún dato a la vista, o incluso controlar la sesión del usuario utilizando algún código especial. ¡**Auth::user()** lo hace todo!

Para que esta página sea accesible, necesitamos añadir una ruta para ella. Vamos a hacer esto en un minuto, pero ahora sería un buen momento para hablar sobre protección de páginas sensibles de nuestra aplicación...

Creando filtros

Laravel 4 incluye un fichero de filtros, en el que podemos definir filtros para ejecutar en rutas simples (o incluso grupos de rutas).

```php
<?php

Route::filter("auth", function()
{
    if (Auth::guest())
    {
        return Redirect::route("user/login");
    }
});

Route::filter("guest", function()
{
    if (Auth::check())
    {
        return Redirect::route("user/profile");
    }
});

Route::filter("csrf", function()
{
    if (Session::token() != Input::get("_token"))
    {
        throw new Illuminate\Session\TokenMismatchException;
    }
});
```

Este fichero debería guardarse como **app/filters.php**.

El primer filtro es para rutas (o páginas si lo prefieres) para las que un usuario debe estar autenticado. El segundo es para todo lo contrario, para las que los usuarios no deben estar autenticados. El último filtro es el que hemos estado utilizando desde el principio.

Cuando usamos el método **Form::open()**, Laravel añade automáticamente un campo oculto en nuestros formularios. Este campo contiene un token especial de seguridad que es comprobado cada vez que el formulario es enviado. No necesitas realmente entender porqué esto es más seguro...

...pero si quieres, lee esto: **http://blog.ircmaxell.com/2013/02/preventing-csrf-attacks.html**

Para poder aplicar estos filtros, necesitamos modificar nuestro fichero de rutas.

```php
<?php

Route::group(["before" => "guest"], function()
{
    Route::any("/", [
        "as"   => "user/login",
        "uses" => "UserController@loginAction"
    ]);

    Route::any("/request", [
        "as"   => "user/request",
        "uses" => "UserController@requestAction"
    ]);

    Route::any("/reset", [
        "as"   => "user/reset",
        "uses" => "UserController@resetAction"
    ]);
});

Route::group(["before" => "auth"], function()
{
    Route::any("/profile", [
        "as"   => "user/profile",
        "uses" => "UserController@profileAction"
    ]);

    Route::any("/logout", [
        "as"   => "user/logout",
        "uses" => "UserController@logoutAction"
    ]);
});
```

Este campo debería guardarse como **app/routes.php**.

Para proteger las partes de nuestra aplicación, juntamos grupos con el método **Route::group()**. El primer argumento nos permite especificar qué filtros aplicar a las rutas encerradas en él. Queremos

agrupar todas nuestras rutas en las que los usuarios no deban estar autenticados, para que esos usuarios no las vean cuando están logados. Hacemos lo contrario para la página de perfil porque solo los usuarios autenticados deberían poder ver sus páginas de perfil.

Creando una acción de salida (logout)

Para probar estas nuevas medidas de seguridad (y redondear el tutorial) necesitamos crear un método **logoutAction()** y añadir enlaces en la cabecera para que los usuarios puedan salir de su sesión.

```
public function logoutAction()
{
    Auth::logout();
    return Redirect::route("user/login");
}
```

Esto se ha extraído de **app/controllers/UserController.php**.

El método **logoutAction()** llama al método **Auth::logout()** para cerrar la sesión del usuario, y dirigirlo de vuelta a la pantalla de acceso. ¡Así de fácil!

Este es el aspecto de lo que incluye la nueva cabecera:

```
@section("header")
    <div class="header">
        <div class="container">
            <h1>Tutorial</h1>
            @if (Auth::check())
                <a href="{{ URL::route("user/logout") }}">
                    logout
                </a>
                |
                <a href="{{ URL::route("user/profile") }}">
                    profile
                </a>
            @else
                <a href="{{ URL::route("user/login") }}">
                    login
                </a>
            @endif
```

```
18          </div>
19      </div>
20  @show
```

Este fichero debería guardarse como **app/views/header.blade.php**.