Laravel nivel intermedio

Route Model Binding

Normalmente cuando se incluye el ID de un registro en la URL, como por ejemplo en una ruta como `/articulos/{id}` de tipo `GET`, lo más común es que el controlador de turno tenga que recuperar de la base de datos el registro con el ID indicado. Es por esto que Laravel permite inyectar directamente el modelo entero en lugar del `id`.

A continuación se muestra un ejemplo en el router:

```php
<?php

use App\Models\Articulo;

Route::get('/articulos/{articulo}', function (Articulo $articulo) {
    return $articulo->titulo;
});
```

Como puede verse, la función en lugar de recibir el ID como parámetro recibe directamente el modelo. Esto ha sido gracias a que el parámetro ha sido declarado del tipo `Articulo` y el nombre del parámentro (`$articulo`) coincide con el parametro de la ruta (`{articulo}`).

Igualmente se puede hacer en el controlador:

```php
<?php

use App\Http\Controllers\ArticuloController;
use App\Models\Articulo;

// Definición del método en el Controlador:
public function show(Articulo $articulo)
{
    return view('articulos.index', ['articulo' => $articulo]);
}
```

```php
<?php

// Definición de la ruta en web.php:
Route::get('/articulos/{articulo}', [ArticuloController::class, 'show']);
```

Hands on!

Actualiza la ruta y el controlador ArticuloController de tu aplicación para que utilice el `Route Model Binding` cuando el usuario solicite los detalles de un artículo (show).

Borrado de registros

El borrado de registros es un tema que suele traer complicaciones, debido a que desde una página HTML solo es posible enviar peticiones `GET` y `POST` (desde formularios). Por lo tanto, las alternativas son las siguientes:

- Crear una ruta de tipo `GET` específica para el borrado. Por ejemplo: `/articulos/destroy/{id}`
- Hacer la petición de tipo DELETE utilizando AJAX y especificando en la llamada el tipo de método: `'type': 'DELETE'`
- Emular la llamada `DELETE` mediante el campo oculto `_method`. Para ello podemos utilizar los helpers o directivas de Laravel en un formulario para notificar que se trata de una petición de tipo `DELETE`:

```
<form method="POST">
      @csrf
      @method("DELETE")

      <button type="submit">Eliminar</button>
</form>
```

Siguiendo con la última de las opciones, quedaría añadir la ruta de borrado al router e implementar el método `destroy()` del controlador:

```
<?php

// Nueva ruta en /router/web.php:
Route::delete('/articulos/{articulo}', [ArticuloController::class, 'destroy'])->name('articulos.destroy');

// Método destroy() en ArticuloContoller:
public function destroy(Articulo $articulo)
{
    $articulo->delete();
    return redirect(route('articulos.index'));
}
```

El borrado de un modelo se puede hacer de forma sencilla invocando al método `delete()` del modelo.

Hands on!

Añade la opción de eliminar cualquier artículo de la aplicación.

Actualizar un modelo

Al igual que ocurre con la creación de un nuevo modelo, para actualizar un modelo los pasos a seguir son los siguientes:

1. Crear una vista que contenga el formulario de actualización. La ruta en este caso será `/articulos/{articulo}/edit`.
2. Crear dos rutas que llamen a los métodos `edit()` y `update()` del controlador: el método `edit()` será el encargado de cargar la vista de actualización (creada en el punto anterior) y el método `update()` recibirá los datos del formulario enviados por el usuario y actualizará el modelo en la base de datos.

Solución

Nueva vista `/resources/views/articulos/edit.blade.php`:

```
<html>
<head>
    <title>RevistApp</title>
</head>
<body>
    <h1>Revistapp</h1>
    <h2>Crear un artículo:</h2>

    <form method="POST" action="{{ route('articulos.update', $articulo) }}">
        @csrf
        @method('PUT')
        <p><label>Titulo: </label><input type="text" name="titulo" value="{{
$articulo->titulo }}"></p>
        <p><label>Contenido: </label><input type="text" name="contenido"
value="{{ $articulo->contenido }}"></p>
        <button type="submit">Crear</button>
    </form>
    <a href="{{ route('articulos.index') }}">Volver</a>

</body>
</html>
```

En el código anterior puede verse cómo se han asignado los valores actuales a los atributos `value` de cada campo. Se ha utilizado la directiva `@method('PUT')` de Laravel para indicar que el método de envío será de tipo `PUT`.

Nuevas rutas añadidas a `/routes/web.php`:

```
<?php

Route::get('/articulos/{articulo}/edit', [ArticuloController::class, 'edit'])-
>name('articulos.edit');
Route::put('/articulos/{articulo}', [ArticuloController::class, 'update'])-
>name('articulos.update');
```

En cuanto a los métodos de `ArticuloController.php`, se utilizará el método `update()` para actualizar el modelo:

```php
<?php
public function edit(Articulo $articulo)
{
    return view('articulos.edit', [
        'articulo' => $articulo
    ]);
}

public function update(Request $request, Articulo $articulo)
{
    $validated = $request->validate([
        'titulo' => 'required|string|max:255',
        'contenido' =>'required|string'
    ]);
    $articulo->update($validated);
    return redirect(route('articulos.show', $articulo->id));
}
```

Construir layouts

Las aplicaciones siempre contienen varias parte de la interfaz que son comunes en todas las páginas (la cabecera, menú de navegación, footer, etc.). Una de las características de Blade es el uso de Layouts, los cuales permiten de forma muy sencilla compartir entre distintas vistas las partes que tienen en común y así evitar repetir lo mismo múltiples veces. La idea consiste en **separar en un archivo distinto la parte común de nuestras vistas** y especificar en ella las zonas que albergarán los contenidos específicos de cada vista (lo que no es común).

Empecemos por definir un layout básico:

```
<html>
    <head>
        <title>App Name - @yield('titulo')</title>
    </head>
    <body>
        <div class="container">
            @yield('content')
        </div>
        <div class="big-footer">
            @yield('footer')
        </div>
    </body>
</html>
```

La directiva `@yield` se utiliza para especificar el lugar donde se mostrarán los contenidos de cada sección.

Ahora crearemos la vista concreta que especificará el contenido a introducir en el layout. Es por esto que decimos que la vista extiende (`extends`) del layout, es decir, la vista heredará toda la estructura definida en el layout y sobreescribirá las partes concretas que defina (las secciones).

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

@section('titulo', 'Page Title')

@section('content')
    <h1>Hello World!</h1>
    <p>This is my body content.</p>
@endsection

@section('footer')
    <p>Built by @JonVadillo.</p>
@endsection
```

`@section` indica la sección del padre donde será introducido el contenido especificado entre las etiquetas `@section` y `@endsection` .

Hands on!

Crea un layout que englobe la parte común que contienen todas las vistas de la aplicación RevistApp. Actualiza las vistas para que extiendan el layout creado.

Laravel Vite: cómo trabajar con código JS y CSS

Introducción

Hoy en día en el desarrollo de frontend moderno se utilizan herramientas que compilan y opmitizan el código Javascript y CSS. En la actualidad predominan Webpack y Vite, siendo este último el que Laravel incluye por defecto a partir de su versión `9.19` .

Desarrollo de frontend con Laravel

Por seguridad la única carpeta accesible desde el navegador es `/public` . Nuestro servidor apunta siempre a la carpeta `/public` , en la que se encuentra el archivo `index.php` encargado de cargar el framework y redireccionar la petición a la ruta correspondiente.

Sería posible incluir nuestros archivos `.css` o `.js` directamente dentro de nuestra carpeta `/public`, pero lo habitual en desarrollo web es realizar algún tipo de compilación: por ejemplo utilizar archivos `.sass` o algún framework de JavaScript. Es por esto que es necesaria una herramienta como Vite o Webpack.

Configuraremos Vite para que compile los archivos `.css` y `.js` que modifiquemos dentro de la carpeta `/resources` y deje el resultado de la compilación en la carpeta `/public`.

Configuración de Vite

Al crear un nuevo proyecto, Laravel crea automáticamente un archivo `vite.config.js` como este:

```
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';

export default defineConfig({
    plugins: [
        laravel({
            input: ['resources/css/app.css', 'resources/js/app.js'],
            refresh: true,
        }),
    ],
});
```

La configuración por defecto ya indica a Vite dónde se ubican los archivos `.css` y `.js`.

Nota importante: si estás trabajando con Laravel Sail desde WSL, es posible que tengas problemas para que el comando `npm run dev` esté pendiente de los cambios en los archivos estáticos. Para que funcione correctamente tendrás que añadir lo siguiente a en el archivo ``vite.config.js`:

```
server: {
    hmr: {
        host: 'localhost',
    },
    watch: {
        usePolling: true,
    },
},
plugins: [
    ...
```

Instalar las dependencias e iniciar el servidor de Vite

Para utilizar Vite es necesario tener Node instalado, ya que utilizaremos NPM para instalar las dependencias. Compruébalo mediante el siguiente comando:

```
node -v
```

En caso de no tener Node instalado, puedes hacerlo descargándolo desde la página web oficial.

A continuación tienes que instalar las dependecias de tu proyecto definidas en el archivo `package.json` ejecutando el comando `npm install`. Las dependencias se instalaran en la carpeta `node_modules`, incluidas Vite y el plugin de Vite para Laravel.

El siguiente paso será iniciar el servidor de Vite:

```
npm run dev
```

El servidor de Vite no tiene nada que ver con el servidor web, se trata de un servidor independiente de Vite encargado de realizar las tareas relacionadas con nuestro frontend.

Referenciar los archivos JS y CSS en las plantillas de Blade

Para incluir nuestros recursos (assets) en cualquier plantilla de Blade utilizaremos la siguiente directiva:

```
@vite(['resources/css/app.css', 'resources/js/app.js'])
```

Publicar los cambios realizados

Una vez terminemos nuestro desarrollo, ya no será necesario nuestro servidor Vite. Lo único que tendremos que hacer será publicar en la carpeta `/public` los archivos compilados y optimizados. Para ello es necesario ejecutar la siguiente sentencia:

```
npm run build
```

La consola nos mostrará la dirección de los archivos publicados (`/public/build/assets/`). Este paso será necesario antes de desplegar nuestra aplicación en producción.

Utilizar Bootstrap en tu proyecto

A diferencia de versiones anteriores, a partir de su versión 6, Laravel no incluye por defecto las dependencias necesarias para Bootstrap. Por lo tanto, tendremos 4 opciones para utilizar Bootstrap:

a) Referenciar las dependecias JS y CSS utilizando BootstrapCDN (enlaces disponibles en la documentación oficial. Tal y como indica la web oficial, bastaría con lo siguiente:

```
<link
href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css"
rel="stylesheet" integrity="sha384-
1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3"
crossorigin="anonymous">b) Descargar las dependecias ([enlace]
(https://getbootstrap.com/docs/4.4/getting-started/download/)) e incluirlas
manualmente en las carpetas `/public/css` y `/public/js`.
...

<script
src="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/js/bootstrap.bundle.min.
js" integrity="sha384-
ka7Sk0Gln4gmtz2MlQnikT1wXgYsOg+OMhuP+IlRH9sENBO0LRn5q+8nbTov4+1p"
crossorigin="anonymous"></script>
```

o bien si tiramos de archivos que están en nuestro proyecto dentro de la carpeta `public` en lugar de un CDN:

```
<link rel="stylesheet" href="{{asset('vendor/css/bootstrap.min.css')}}">
...
<script src="{{asset('vendor/js/bootstrap.bundle.min.js')}}"></script>
```

c) Utilizar Laravel Mix para compilar nuestros archivos JS y CSS.

> Laravel ha sustituido Laravel Mix por Vite a partir de la versión `9.19`, por lo que Vite es la opción más recomendada.

d) Utilizar Vite como herramienta de compilación. Actualmente es la opción por defecto de Laravel.

Bootstrap con Vite

Como Laravel no usa Sass por defecto, cambiaremos el nombre de nuestro directorio `css` por `scss` y el archivo incluido `app.css` por `app.scss` para seguir esa convención.

Después de eso, tenemos que actualizar nuestro `vite.config.js` file para utilizar la nueva ruta en su `input` array:

```
// vite.config.js
export default defineConfig({
    plugins: [
        laravel({
            input: ['resources/scss/app.scss', 'resources/js/app.js'],
            refresh: true,
        }),
    ]
});
```

Lo siguiente que vamos a hacer es crear un alias para Bootstrap en nuestro fichero `vite.config.js` para que sea más sencillo de importar:

```
...
import path from 'path';

export default defineConfig({
    plugins: [ /...
                .../ ],
    resolve: {
        alias: {
            '~bootstrap': path.resolve(__dirname, 'node_modules/bootstrap')
        }
    }
});
```

Con nuestro alias configurado, lo que tenemos que hacer ahora es importar el CSS de Bootstrap en nuestro archivo Sass:

```
// resources/scss/app.scss
@import "~bootstrap/scss/bootstrap";
```

Esto importará **todas** las hojas de estilo que ofrece Bootstrap.

Con eso hecho, lo único que queda es añadir nuestra directiva Vite de Blade en la vista o layout en la que queremos que se apliquen y donde ya podríamos usar cualquier componente de

Bootstrap:

```
<head>
    @vite(['resources/scss/app.scss', 'resources/js/app.js'])
</head>
<body>
    <header class="p-3 mb-3 border-bottom">
        <div class="container">
            <div class="d-flex flex-wrap align-items-center justify-content-
center justify-content-lg-start">
                <a href="/" class="d-flex align-items-center mb-2 mb-lg-0 text-
dark text-decoration-none">
                    <svg fill="none" viewBox="0 0 24 24" stroke-width="1.5"
stroke="currentColor" class="bi me-2" width="40" height="32" role="img" aria-
label="Storefront">
                        <path stroke-linecap="round" stroke-linejoin="round"
d="M13.5 21v-7.5a.75.75 0 01.75-.75h3a.75.75 0 01.75.75V21m-4.5 0H2.36m11.14
0H18m0 0h3.64m-1.39 0V9.349m-16.5 11.65V9.35m0 0a3.001 3.001 0
003.75-.615A2.993 2.993 0 009.75 9.75c.896 0 1.7-.393 2.25-1.016a2.993 2.993 0
002.25 1.016c.896 0 1.7-.393 2.25-1.016a3.001 3.001 0 003.75.614m-16.5 0a3.004
3.004 0 01-.621-4.72L4.318 3.44A1.5 1.5 0 015.378 3h13.243a1.5 1.5 0
011.06.44l1.19 1.189a3 3 0 01-.621 4.72m-13.5 8.65h3.75a.75.75 0
00.75-.75V13.5a.75.75 0 00-.75-.75H6.75a.75.75 0 00-.75.75v3.75c0
.415.336.75.75.75z" />
                    </svg>
                </a>

                <ul class="nav col-12 col-lg-auto me-lg-auto mb-2 justify-
content-center mb-md-0">
                    <li><a href="#" class="nav-link px-2 link-
secondary">Overview</a></li>
                    <li><a href="#" class="nav-link px-2 link-
dark">Inventory</a></li>
                    <li><a href="#" class="nav-link px-2 link-
dark">Customers</a></li>
                    <li><a href="#" class="nav-link px-2 link-
dark">Products</a></li>
                </ul>

                <form class="col-12 col-lg-auto mb-3 mb-lg-0 me-lg-3"
role="search">
                    <input type="search" class="form-control"
placeholder="Search..." aria-label="Search">
                </form>

                <div class="dropdown text-end">
                    <a href="#" class="d-block link-dark text-decoration-none
dropdown-toggle"
                        data-bs-toggle="dropdown" aria-expanded="false">
```

```
                        <img
src="https://avatars.githubusercontent.com/u/98683" alt="mdo" class="rounded-
circle" width="32"
                            height="32">
                    </a>
                    <ul class="dropdown-menu text-small" style="">
                        <li><a class="dropdown-item" href="#">New project...
</a></li>
                        <li><a class="dropdown-item" href="#">Settings</a></li>
                        <li><a class="dropdown-item" href="#">Profile</a></li>
                        <li>
                            <hr class="dropdown-divider">
                        </li>
                        <li><a class="dropdown-item" href="#">Sign out</a></li>
                    </ul>
                </div>
            </div>
        </div>
    </header>
</body>
```

Ahora tenemos que compilar los recursos de nuestra web:

```
npm run build
```

Y así deberíamos ver nuestras nuevas cabeceras de Bootstrap cuando entremos en nuestro site.

Hands on!

Añade estilo a la aplicación mediante el framework Bootstrap 5.

Relaciones One-to-Many

Definir una relación

Las relaciones en Eloquent ORM se definen como métodos. Supongamos que tenemos dos entidades, `User` y `Articulo`. Podríamos decir que un `User` tiene (*has*) varios `Articulo` o que un `Articulo` pertenece a (*belongs to*) un `User`. Por lo tanto, podemos definir la relación en cualquiera de los dos modelos, incluso en los dos.

```php
<?php

class User extends Model
{
    /**
     * Get the articulos records associated with the user.
     */
    public function articulos()
    {
        return $this->hasMany(Articulo::class);
    }
}
```

```php
<?php

class Articulo extends Model
{
    /**
     * Get the user record associated with the articulo.
     */
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

Tienes toda la información sobre cómo definir relaciones entre modelos en la documentación oficial.

Acceder a los modelos de una relación

El acceso se podrá hacer como propiedades del propio modelo, es decir, mediante `$user->articulos` o `$articulo->user`. Esto es gracias a que Eloquent utiliza lo que conocemos como 'dynamic properties' y acceder a los métodos de las relaciones como si fuesen propiedades:

```php
<?php

$user = App\Models\User::find(1); // Ejecuta la sentencia: select * from users
where id = 1

$user_articulos = $user->articulos; // Ejecutara la sentencia: select * from
articulos where user_id = 1

foreach ($user_articulos as $articulo) {
    //
}
```

En el ejemplo anterior, la variable `$user_articulos` contiene una colección de objetos de la clase `Articulo`.

Crear la restricción de las claves foráneas en la Base de Datos

Como es lógico, para que el modelo pueda acceder a otro modelo con el que mantiene una relación one-to-many, es necesario especificar la `foreign key` correspondiente a nivel de base de datos. Recordemos que las foreign key permiten mantener la integridad referencial en nuestra base de datos.

Por defecto, si no indicamos lo contrario, el modelo de Eloquent utilizará como foreign key el nombre del modelo que contiene la colección añadiendo el sufijo `'_id'`. Es decir, en el caso anterior la tabla de `Articulo` deberá contener una columna llamada `'user_id'`, ya que el nombre del otro modelo es `User`. Por lo tanto nuestra migración debería quedar de la siguiente forma:

```php
    public function up()
    {
        Schema::create('articulos', function (Blueprint $table) {
            $table->id();
            $table->foreignId('user_id')->constrained()->cascadeOnDelete();
            $table->string('titulo');
            $table->text('contenido');
            $table->boolean('publicado')
            $table->timestamps();
        });
    }
}
```

Expliquemos la siguiente sentencia:

```
$table->foreignId('user_id')->constrained()->cascadeOnDelete();
```

- `foreignId` crea una columna del tipo `UNSIGNED BIGINT` con el nombre especificado.
- `constrained` utilizará las convenciones de Laravel para determinar la tabla y columna a la que se refiere. Nota: si no siguiésemos las convenciones, podríamos indicarle el nombre de la tabla pasándoselo como argumento: `constrained('users')`.
- `cascadeOnDelete` indica las acciones a realizar cuando se vaya a borrar el registro. El borrado en cascada determina que si el usuario que contiene los artículos es borrado, se borrarán también todos sus artículos. Otras opciones serían `restrictedOnDelete` (restringe el borrado mientras tenga artículos referenciados) o `nullOnDelete` (establece el valor `NULL` a la foreign key de los artículos relacionados).

Consejo: cómo añadir columnas a modelo existente

Existen dos escenarios posibles en los cuales queremos realizar cambios sobre modelos existentes:

- Estamos desarrollando una nueva aplicación y no nos importa borrar los datos existentes.
- Tenemos una aplicación en uso y queremos añadir columnas sin perder ningún registro.

En el primer caso, es suficiente con modificar la migración de la tabla correspondiente y ejecutar el comando:

```
php artisan migrate:fresh
```

Este comando eliminará todas las tablas de la base de datos y volverá a crearlas desde cero.

Para el segundo caso (modificar una tabla sin perder datos), lo recomendable es crear una nueva migración y ejecutar el comando `php artisan migrate` para lanzar los cambios. Normalmente se incluyen los cambios en el propio nombre de la migración, por ejemplo:

```
php artisan migrate:make add_category_to_articulos --table="articulos"
```

```php
<?php

    public function up()
    {
        Schema::table('articulos', function($table)
        {
            $table->string('category');
        });
    }
    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('articulos', function ($table) {
        $table->dropColumn('category');
        });
    }
}
```

Hands on!

La vista de detalle de artículo mostrará los comentarios del artículo e incluirá la posibilidad de añadir nuevos comentarios.

Solución

Crear la migración para los comentarios e implementar el método `up()`. Deberá incluir un campo de texto para almacenar el comentario y la clave foránea indicando el artículo al que pertenece el comentario:

```php
<?php

...

public function up()
{
    Schema::create('comentarios', function (Blueprint $table) {
        $table->id();
        $table->foreignId('articulo_id')->constrained()->cascadeOnDelete();
        $table->text('texto');
        $table->timestamps();
    });
}
...
```

Crear el modelo `/App/Models/Comentario.php` :

```php
<?php

namespace App\Models;
use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Comentario extends Model
{
    use HasFactory;
    protected $fillable = [
        'texto',
    ];

    public function articulo()
    {
        return $this->belongsTo(Articulo::class);
    }

}
```

Añadir las relación en la clase Articulo:

`/App/Models/Articulo.php` :

```php
<?php

class Articulo extends Model
{
    ...

    public function comentarios()
    {
        return $this->hasMany(Comentario::class);
    }
}
```

Actualizar la vista para que muestre el formulario de creación de un comentario y el listado de comentarios del artículo:

`show.blade.php`:

```blade
<!DOCTYPE html>
<html lang="es">
<head>
    <title>RevistApp</title>
</head>
<body>
    <h1>Revistapp</h1>
    <h2>Detalle del artículo:</h2>
    <ul>
        <li>Fecha de creación: {{ $articulo->created_at }}</li>
        <li>Fecha de última actualización: {{ $articulo->updated_at }}</li>
        <li>Titulo: {{ $articulo->titulo }}</li>
        <li>Contenido: {{ $articulo->contenido }}</li>
    </ul>
    <h3>Añadir comentario</h3>
    <form method="POST" action="{{ route('comentarios.store', $articulo) }}">
        @csrf
        <div>
            <label>Contenido:</label>
        </div>
        <div>
            <textarea name="texto"></textarea>
        </div>
        <div>
            <button type="submit">Crear</button>
        </div>
    </form>
    <h3>Comentarios:</h3>
    <ul>
    @foreach ($articulo->comentarios as $comentario)
        <li>
            <small>{{ $comentario->created_at }}</small>:
            <span>{{ $comentario->texto }}</span>
        </li>
    @endforeach
    </ul>

    <a href="{{ route('articulos.index') }}">Volver</a>
</body>
</html>
```

Crear el nuevo controlador `ComentarioController.php` encargado de la creación de comentarios:

```php
<?php

namespace App\Http\Controllers;

use App\Models\Comentario;
use App\Models\Articulo;
use Illuminate\Http\Request;

class ComentarioController extends Controller
{

    public function store(Request $request, Articulo $articulo)
    {
        $comentario = new Comentario;

        $comentario->texto = $request->texto;
        $articulo->comentarios()->save($comentario);

        /* Alternativa:
        $comentario->articulo_id = $articulo->id;
        $comentario->save();
        */

        return redirect()->route('articulos.show', $articulo->id);
    }
}
```

Registrar la nueva ruta en el router `web.php`:

```
Route::post('articulos/{articulo}/comentarios', [ComentarioController::class,
'store'])->name('comentarios.store');
```

Generar datos de prueba

Laravel incluye un mecanismo llamado Seeder que sirve para rellenar la base de datos con datos de prueba. Por defecto nos incluye la clase DatabaseSeeder en la que podemos incluir el código que genere los datos de prueba:

```php
<?php

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        $faker = \Faker\Factory::create();

        for($i=0;$i<10;$i++){
            DB::table('articulos')->insert([
                'titulo' => $faker->text(50),
                'contenido' => $faker->text(400)
            ]);
        }
    }
}
```

Una vez creado nuestro Seeder es probable que necesites regenerar el fichero autoload.php:

```
composer dump-autoload
```

Por último, sólo nos quedaría lanzar el proceso de 'seeding':

```
php artisan db:seed
```

Si lo que quieres es lanzar el proceso de creación de base de datos y el de seeding a la vez, puedes utilizar el siguiente comando:

```
php artisan migrate:fresh --seed
```

Generar Seeders específicos

Es recomendable crear un seeder específico por cada entidad. Para ello, puedes utilizar el siguiente comando:

```
php artisan make:seeder ArticuloSeeder
```

De forma que tengamos:

```
class ArticuloSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        $faker = Factory::create();
        for ($i = 0; $i < 50 ; $i++) {
            Articulo::create([
                'titulo'=>$faker->sentence,
                'contenido'=>$faker->text,
                'publicado'=>$faker->boolean
            ]);
        }
    }
}
```

Por último, tendrás que modificar la clase DatabaseSeeder para que lance nuestros Seeders:

```
<?php

public function run()
{
     $this->call([
        ArticuloSeeder::class
        //Y los que tuvieramos
     ]);
}
```

Y volver a lanzar el comando:

```
php artisan db:seed
//o bien
php artisan migrate:fresh --seed
```

Uso de factories

A la hora de generar datos para testing también es recomendable utilizar Factories para crear los objetos de BBDD. Una Factory simplemente es una clase que define los atributos que tendrá un objeto en su creación. En el siguiente ejemplo se muestra una posible factory para la clase Articulo:

```php
<?php

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

class ArticuloFactory extends Factory
{
    /**
     * Define the model's default state.
     *
     * @return array
     */
    public function definition()
    {
        return [
            'titulo' => $this->faker->text($maxNbChars = 50),
            'contenido' => $this->faker->text($maxNbChars = 400),
            'publicado' => $this->faker->boolean
        ];
    }
}
```

Es suficiente con implementar el método `definition` y especificar en él las propiedades del objeto que se creará. Una vez tenemos creadas las factories, solo nos quedaría utilizarlas desde el Seeder correspondiente:

```php
<?php

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        \App\Models\Articulo::factory(20)->create();
    }
}
```

La creación de factories se puede realizar mediante el comando `php artisan make:factory`. Por ejemplo:

```
php artisan make:factory ArticuloFactory
```

Autenticación

La autenticación es una funcionalidad presente en la gran mayoría de aplicaciones. Básicamente se trata de asegurar que un usuario es quién dice ser mediante un control de acceso a la aplicación.

Las opciones principales que provee Laravel para implementar la autenticación son:

- Laravel UI
- Laravel Breeze (opción recomendada a partir de Laravel 8)

A pesar de que la opción recomendada para crear la estructura inicial sea Laravel Breeze, la forma en la que accederemos a la información del usuario autenticado o el modo de securizar las rutas será el mismo.

Laravel UI

!!! warning inline end "Importante"

```
La opción que se recomienda actualmente para implementar la autenticación es Lara
```

A partir de la versión 6 de Laravel es posible utilizar el paquete `laravel/ui` para implementar funcionalidades de autenticación. Laravel UI nos trae de serie algunos elementos necesarios para implementar la autenticación en nuestras aplicaciones y no tener que preocuparnos de hacer todas las tareas por nosotros mismos (login, registro, recuperación de contraseña, validación de usuario, etc.).

En concreto necesitaremos lo siguiente:

- Generar las vistas (login, registro, etc.), rutas y sus respectivas implementaciones.
- Especificar las partes de nuestra web (rutas) que queramos proteger.

Crear la estructura necesaria con Laravel UI

El primer paso es instalar el paquete de laravel/ui mediante Composer:

```
composer require laravel/ui
```

Este paquete nos permitirá generar de forma automática todo lo relacionado con la interfaz de usuario: vistas, rutas y un nuevo controlador llamado `HomeController`. Tail y como dice la documentación oficial, en función de nuestras necesidades, podremos elegir entre uno de los siguientes comandos:

```
// Generate basic scaffolding...
php artisan ui bootstrap
php artisan ui vue
php artisan ui react

// Generate login / registration scaffolding...
php artisan ui bootstrap --auth
php artisan ui vue --auth
php artisan ui react --auth
```

En el caso de esta guía ejecutaremos el comando que genera las funciones de login y registro utilizando el framework Bootstrap:

```
php artisan ui bootstrap --auth
```

Si te fijas bien, el router `web.php` incluye dos nuevas líneas:

```
Auth::routes();

Route::get('/home', 'HomeController@index')->name('home');
```

La primera genera de forma automática las rutas necesarias para el proceso de autenticación (prueba a acceder a `/login` o `/register` para comprobarlo). Las vistas que se cargan son las autogeneradas en `resources/views/auth` (también se ha creado un layout).

La segunda es simplemente un nuevo controlador autogenerado como ejemplo. Si intentas acceder a la ruta creada `/home` verás como la aplicación nos lleva a la página de login. Esto es porque el controlador `HomeController` está definido como 'seguro' y solo usuarios autenticados podrán acceder.

¡Felicidades! Ya tienes la estructura básica de la aplicación creada. Puedes probar a registrar nuevos usuarios y realizar las acciones de login o logout con ellos.

Configuración básica

Una de las cosas que tendrás que configurar es la ruta a la que se envía al usuario tras autenticarse. Esto puede especificarse mediante la variable `HOME` del archivo `RouteServiceProvider.php`:

```
public const HOME = '/home';
```

Laravel utiliza por defecto el campo `email` para identificar a los usuarios. Puedes cambiar esto creando un método `username()` en el controlador `LoginController.php`.

```
<?php

public function username()
{
    return 'username';
}
```

Otro aspecto que podremos configurar es la ruta a la que enviaremos al usuario cuando intente acceder a una ruta protegida sin autenticarse. Por defecto Laravel le enviará a `/login`, pero podemos cambiar esto modificando el método `redirectTo()` del archivo `app/Http/Middleware/Authenticate.php`:

```php
<?php

protected function redirectTo($request)
{
    return route('login');
}
```

Laravel Breeze

A partir de la versión 8 de Laravel se recomienda utilizar Laravel Breeze, el cual utiliza Tailwind CSS en lugar de Bootstrap. Este aspecto es importante ya que afecta a las vistas creadas. Laravel Breeze es una implementación sencilla de las funciones más habituales de autenticación como: login, registro, recuperación de contraseña, verificación de correo electrónico o confirmación de contraseña por correo. Para ello creará todas las vistas, rutas y controladores necesarios y además los dejará disponibles en el código de nuestro proyecto para que podamos modificar todo aquello que necesitemos.

> Laravel Breeze debe instalarse tras la creación del proyecto". Laravel Breeze debe instalarse sobre un proyecto recién creado de Laravel, ya que eliminará código existente en rutas, etc. Es lo que Laravel considera un Starter Kit.

Crear la estructura necesaria con Laravel Breeze

Para instalar Laravel Breeze es necesario lanzar el siguiente comando, el cual instalará el paquete utilizando Composer:

```
composer require laravel/breeze --dev
```

A continuación el siguiente comando generará todo el código necesario en tu proyecto:

```
php artisan breeze:install
```

El comando anterior habra hecho varios cambios en el proyecto, como por ejemplo:

- Crear los controladores necesarios para el login, registro, recuperación de contraseña, etc.
- Crear las vistas empleadas por los controladores (utilizando Tailwind CSS).
- Crear una vista llamada Dashboard que utilizaremos cuando un usuario se autentica correctamente. Para ello también actualiza la variable `HOME` del archivo `RouteServiceProvider.php`.

- Registrar la ruta `/dashboard` y cargar el archivo de rutas `auth.php`, el cual incluye las rutas necesarias para el login, registro, etc.
- Crear los ficheros CSS y JS necesarios, que luego habrá que compilar. También añade al archivo `package.json` dependencias como TailwindCSS o AlpineJS.
- Crear las rutas relacionadas con la autenticación en el archivo `auth.php`.

Al igual que se hace con el resto de archivos estáticos que queremos compilar y publicar en la carpeta `/public`, tendremos que lanzar los comandos `npm install` y `npm run dev`.

Por último, no olvides lanzar las migraciones necesarias mediante el comando `php artisan migrate`.

Securizar rutas

Indicaremos las rutas que queramos proteger directamente en nuestro ruter `web.php`:

```
<?php

Route::get('profile', function () {
    // Solo podrán acceder usuarios autenticados.
})->middleware('auth');

// Otro ejemplo:
Route::get('articulos', [ArticuloController::class, 'index'])-
>name('articulos.index')->middleware('auth');
```

También podremos indicarlo directamente en el constructor de un controlador de la siguiente forma:

```
<?php

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

Acceder al usuario autenticado

Existen distintas formas de acceder al objeto del usuario autenticado. Desde cualquier punto de la aplicación podremos acceder utilizado la facade `Auth`:

```php
<?php

use Illuminate\Support\Facades\Auth;

// Conseguir el usuario autenticado:
$user = Auth::user();

// Conseguir el ID del usuario autenticado:
$id = Auth::id();
```

También podremos conseguirlo desde cualquier petición que reciba nuestro controlador:

```php
<?php

public function update(Request $request)
{
    $request->user(); //  devuelve una instancia del usuario autenticado.
}
```

Para comprobar si un usuario está autenticado, podemos emplear el método `check()`:

```php
<?php

use Illuminate\Support\Facades\Auth;

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

Redireccionar al usuario no autenticado a una página

Cuando nuestro middleware de autenticación detecte que un usuario tiene que está autenticado para acceder a una ruta, automáticamente redirigirá al usuario donde nosotros le indiquemos (habitualmente una vista de login). Esto lo podremos indicar en el método `redirectTo()` del archivo `app/Http/Middleware/Authenticate.php`.

```php
<?php

protected function redirectTo($request)
{
    return route('login');
}
```

Hands on!

- Añade las funciones de login, registro y logout a la aplicación.
- Protege la ruta empleada para escribir un nuevo artículo (solo usuarios autenticados podrán acceder).
- La opción de borrar un artículo únicamente estará visible para usuarios autenticados.
- Muestra los comentarios de los artículos únicamente a usuarios autenticados. A los usuarios no identificados muéstrales un mensaje con un enlace a la página de login.

Autenticación en Blade

También podremos comprobar en nuestras vistas si un usuario está autenticado o no. Para ello tendremos la directiva `@auth` y `@guest`:

```
@auth
    // Si el usuario está autenticado...
@endauth

@guest
    // Si el usuario no está autenticado...
@endguest
```

Autorización

Muchas veces la autenticación no es suficiente y queremos especificar **qué acciones puede realizar cada usuario sobre una serie de recursos determinados**. Por ejemplo: ¿Puede cualquier usuario autenticado insertar nuevos artículos en una aplicación, o únicamente debería poder hacerlo un usuario de tipo administrador?¿Quién puede modificar un artículo, cualquier usuario o únicamente su creador?

Aparte de proveernos de mecanismos para la autenticación, Laravel también facilita la autorización de las acciones que pueden realizar los usuarios. Existen dos formas de gestionar la autorización:

- **Gates**: Hacen referencia a los "permisos" tal y como los conocemos. Ejemplos de gates pueden ser: "crear_usuario", "editar_articulo", etc. En función de si un usuario tiene un permiso determinado, podrá por ejemplo visualizar una parte de nuestra vista o ejecutar una acción en un controlador.
- **Policies**: Se utilizan cuando queremos definir permisos sobre modelos concretos. Es decir, agrupan las reglas de autorización sobre un modelo concreto.

A la hora de definir las reglas de autorización de la aplicación, podrás elegir entre utilizar gates, policies o una combinación de las dos. En esta guía únicamente se abordará el uso de las gates.

Autorización mediante Gates

Los pasos a seguir son siempre los siguientes:

1. Definir el tipo de permiso o `Gate`.
2. Comprobar el permiso en el front-end (por ejemplo mostrar u ocultar un contenido) o en el back-end (por ejemplo comprobar si el usuario puede insertar un dato).

Definir un Gate

Los Gates se tiene que definir en el método `boot()` de la clase `App\Providers\AuthServiceProvider.php`. Vemos un ejemplo donde se definen dos gates:

```php
<?php

use App\Models\Post;
use App\Models\User;
use Illuminate\Support\Facades\Gate;

/**
 * Register any authentication / authorization services.
 *
 * @return void
 */
public function boot()
{
    $this->registerPolicies();

    // Devuelve TRUE si el usuario tiene el valor 'is_admin' a 1.
    Gate::define('manage_users', function(User $user) {
        return $user->is_admin == 1;
    });

    // Devuelve TRUE si el usuario es el creador del Articulo.
    Gate::define('update-articulo', function (User $user, Articulo $articulo) {
        return $user->id === $articulo->user_id;
    });
}
```

Explicación de las dos Gates anteriores:

- `'manage_users'`: esta Gate comprobará que el usuario es de tipo administrador, es decir, que tiene el atributo `is_admin` con el valor 1. Devolverá `true` si se cumple la condición.
- `'update-articulo'`: esta Gate comprobará que el creador del Articulo es el usuario actual. Para ello comprobará que el `id` del usuario sea el mismo que el del creador del Articulo (atributo `user_id`).

Comprobar un permiso en el front-end

Tendremos disponible la directiva `@can` o `@cannot` en Blade:

```
<ul>
    <li>
        <a href="{{ route('articulos.index') }}">Articulos</a>
    </li>
    @can('manage_users')
    <li>
        <a href="{{ route('users.index') }}">Usuarios</a>
    </li>
    @endcan
</ul>
```

O por ejemplo:

```
@can('update-articulo', $post)
    <!-- El usuario actual puede actualizar el artículo... -->
    <li>
        <a href="{{ route('articulos.edit') }}">Editar</a>
    </li>
@elsecan('create-articulo', App\Models\Post::class)
    <!-- El usuario actual NO puede actualizar el artículo... -->
    <li>
        <a href="{{ route('articulos.create') }}">Editar</a>
    </li>
@else
    <!-- El usuario actual NO puede actualizar el artículo... -->
    <li>
        <span>No puedes crear ni editar</span>
    </li>
@endcan
```

Comprobar un permiso en el Controlador

Laravel habilita la facade `Gate` para que podamos utilizarla desde nuestros controladores:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Articulo;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Gate;

class ArticuloController extends Controller
{
    /**
     * Update the given articulo.
     *
     * @param  \Illuminate\Http\Request  $request
     * @param  \App\Models\Articulo  $articulo
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, Articulo $articulo)
    {
        if (! Gate::allows('update-article', $articulo)) {
            abort(403);
        }

        // Actualizar el articulo...
    }
}
```

Es posible realizar comprobaciones de varios permisos mediante los métodos `any()` o `none()`:

```php
<?php

if (Gate::any(['update-articulo', 'delete-articulo'], $post)) {
    // El usuario puede actualizar o borrar el articulo...
}

if (Gate::none(['update-articulo', 'delete-articulo'], $post)) {
    // El usuario no puede actualizar o borrar el artículo...
}
```

También tendremos disponibles los métodos `can()` y `cannot()` en el objeto del usuario actual:

```php
<?php

public function store(Request $request)
{
    if (!$request->user()->can('create_users'))
        abort(403);
    }
}
```

```php
<?php

public function store(Request $request)
{
    if ($request->user()->cannot('create_users'))
        abort(403);
    }
}
```

Si no tienes el objeto `$request` puedes utilizar también el método `auth()` para acceder al objeto del usuario autenticado:

```php
<?php

public function create()
{
    if (!auth()->user()->can('create_users'))
        abort(403);
    }
}
```

Comprobar un permiso en el Router

Otro forma de realizar la autorización es indicándola como middleware en la ruta:

```php
<?php

Route::post('users', [UserController::class, 'store'])
    ->middleware('can:create_users');
```

El formato siempre será el mismo: `can:xxxxx` o `cannot:xxxxx`.

Tienes más información sobre la autorización en la documentación oficial de Laravel.

Manejo de sesiones

HTTP es un protocolo sin estado (stateless), es decir, no guarda ninguna información sobre conexiones anteriores. Esto quiere decir que nuestra aplicación no tiene "memoria", y cada petición realizada por un usuario es nueva para la aplicación. Las sesiones permiten afrontar este problema, ya que son un mecanismo para almacenar información entre las peticiones que realiza un usuario al navegar por nuestra aplicación. Laravel implementa las sesiones de forma que su uso es muy sencillo para los desarrolladores.

Configuración

Laravel soporta el manejo de sesiones con distintos backends (bases de datos, ficheros, etc.). Esta configuración se indica en el fichero `config/session.php`, en le que podemos indicar el driver a utilizar ("file", "cookie", "database", "apc", "memcached", "redis", "dynamodb" o "array"). La opción utilizada por defecto es "cookie", la cual es suficiente para la mayoría de aplicaciones.

Más información sobre la configuración en la documentación oficial.

Uso de las sesiones

Existen dos formas principales de acceder a la información de la sesión de usuario:

- El helper global `session`:

```php
<?php

// Obtener un valor de la sesión
$value = session('key');

// Podemos indicar un valor por defecto
$value = session('key', 'default');

// Para almacenar un valor, le pasamos un Array:
session(['key' => 'value']);
```

- Mediante la instancia `Request` (inyectada en los métodos de nuestros controladores)

```php
<?php

public function show(Request $request, $id)
{
    $value = $request->session()->get('key');

    // También es posible indicar un valor por defecto si no existe ninguno:
    $value = $request->session()->get('key', 'default');

    // Almacenar un valor
    $request->session()->put('key', 'value');

    // Recuperar un valor y eliminarlo de la sesión
    $value = $request->session()->pull('key', 'default');
}
```

También es posible acceder a valores de la sesión desde las vistas de Blade utilizardo la función `get()`:

```
Session::get('key')
```

Hands on!

Añade las siguientes funcionalidades a la aplicación:

- Guardar en sesión los artículos leídos: cuando un usuario entre a ver un artículo, se almacenará en sesión que ya lo ha leído.
- Guardar en sesión los artículos favoritos de un usuario: el usuario podrá hacer click en un enlace/botón que guarde en sesión ese artículo como favorito.

Los artículos marcados como favoritos se podrán distinguir visualmente (mediante un icono, texto en negrita o similar). Igualmente, los artículos leídos se mostrarán también de forma especial.

Validación de formularios

Realizar la validación de los campos del formulario

Laravel permite validar cualquier campo enviado por un formulario mediante el método `validate`. Tal y como ya habíamos hecho anteriormente:

```php
<?php

public function store(Request $request)
{
    //Validar la petición:
    $validated = $request->validate([
        'titulo' => 'required|string|max:255',
        'contenido' =>'required|string'
    ]);

    Articulo::create($validated);

    return redirect(route('articulos.index'));
}
```

Si la validación pasa correctamente el código seguirá ejecutándose de forma normal y corriente. Pero si la validación falla, se redirigirá al usuario a la página desde la que se ha realizado el envío del formulario.

Puedes ver todas las reglas de validación disponibles aquí.

Mostrar los errores en la vista

Todas las vistas de Laravel tienen disponible la variable llamada `$errors`. En el siguiente ejemplo puede verse cómo mostrar al usuario todos los errores detectados en la validación:

```
<h1>Crear artículo</h1>

@if ($errors->any())
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

Directiva `@error`

La directiva `@error` permite comprobar si un campo concreto ha tenido algún error, y en caso afirmativo mostrar el mensaje de error de dicho campo. Se utilizará de la siguiente manera:

```
<p>
    <label>Titulo: </label>
    <input type="text" name="titulo">
    @error('titulo')
    <small style="color:red;">{{ $message }}</small>
    @enderror
</p>
<p>
    <label>Titulo: </label>
    <input type="text" name="contenido">
    @error('contenido')
    <small style="color:red;">{{ $message }}</small>
    @enderror
</p>
```

En caso de error se mostrará el contenido indicado entre las etiquetas `@error` y `@enderror`. Además la variable `$message` estará disponible entre dichas etiquetas e incluirá el mensaje de error.

Mantener el valor de los campos correctos

En caso de que el formulario tenga varios campos correctos, puede ser interesante mantener los valores enviados previamente en lugar de resetear el formulario entero. El valor de los campos que habían sido completados correctamente puede recuperarse mediante la función `old` de Laravel:

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

Traducción de los mensajes de error

Los mensajes de error pueden verse en el fichero `lang/en/validation.php`. En caso de quere traducirlos a otro idioma, bastaría con crear un fichero con la misma estructura bajo el directorio del nuevo idioma. Por ejemplo: `lang/es/validation.php`

En caso de querer profundizar más en la detección y visualización de errores, puedes encontrar más información en la página oficial.