TRABAJO FIN DE CICLO

INGENIERO DE LA CIERVA
2º DE DESAROLLO DE APLICACIONES WEB

GTV

JORGE ORENES RUBIO
FRAN ARCE CODINA
FULGENCIO VALERA ALONSO

Índice

Presentación

Gtv es una aplicación con la intención de dar información sobre los sitios que te rodean en un momento. El usuario podrá a parte de ver todos los puntos de interés que haya a su alrededor, podrá obtener información sobre ese punto de interés, ver fotos o vídeos. El mismo podrá compartir algún sitio que le resulte interesante.
En este trabajo final de ciclo hemos desarrollado la parte de administración con un framework de php llamado **Laravel** que a su vez incluye **Javascript** , **jQuery**, **Ajax** y estilos de **bootstrap 4**.
En la parte de administración, se podrá ver todos los crud (create, index, update, and delete) como de usuarios, lugares, puntos de interés, fotografías, visitas, etc.
Además, utilizamos roles para los permisos de las distintas clases, de modo que según el rol que tenga un usuario, podrá o no, realizar una serie de acciones. Solo los administradores podrán hacer cualquier acción.

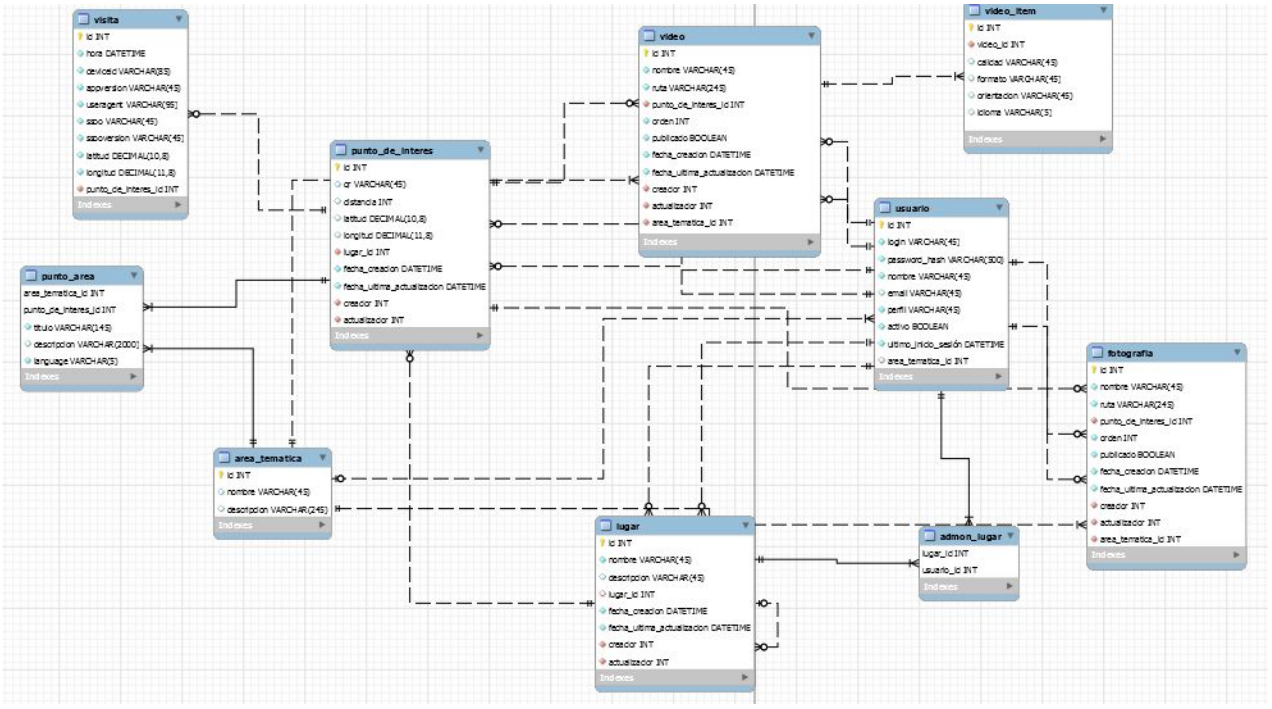
Librerías

Durante el desarrollo de la aplicación, hemos incluido, utilizado y aplicado una serie de librerías:

- **Backend**
  - **spatie/laravel-permissions** → Es una librería de laravel que permite manejar permisos y roles para nuestros usuarios dentro de una aplicación.
  - **simplesoftwareio/simple-qrcode →** Es una libreria que consiste en crear y generar un icono qr de escaneo, con el fin de obtener un código qr desde un móvil.
  - **laravel/ui →** Es una librería de laravel para obtener el login y el registro por defecto, con frontend incluido.
  - **Laravel Charts** → Es una librería de gráficos para laravel que puede generar combinaciones ilimitadas de gráficos listos para usar. Esto se debe a que la API de Chart está diseñada para ser extensible y personalizable, permitiendo cualquier opción en la biblioteca de JavaScript para ser usado rápidamente sin esfuerzo.
  - **vinkla/hashids** → Es una librería que permite ocultar un id o cualquier número a la vista de los usuarios.
- **Frontend**
  - **Croppie.js →** Es una librería JQuery ya que proporciona un plugin basado en ajustar el tamaño de una foto antes de subirlo y enviarlo al servidor.
  - **Sweetalert2 →** Es otra de las librerías de JQuery que permite mostrar una notificación en forma de alerta. Por ejemplo, cuando queremos borrar algo o cerrar sesión.
  - **Datatables.js →** Es una librería JQuery que nos permite pintar tablas con paginación, búsqueda, orden por columnas, etc.
  - **jasny-bootstrap →** Es una librería de Bootstrap que proporciona estilos basado en un campo de un formulario de subida de archivos, sobre todo, en imágenes.
  - **selectpicker →** Librería de frontend que mejora los estilos del campo select.
  - **datetimepicker →** Librería de bootstrap y JQuery ya que a la hora de pinchar un campo, aparezca una fecha u hora a seleccionar.

Base de datos

La base de datos se compone de 10 tablas , dos de ellas tablas pivotes y cinco de ellas internas dedicado a roles y permisos, que los crearemos después de instalar la librería **laravel-permissions**.



- Tabla **usuarios**: en esta tabla se registra los usuarios , independientemente del rol,con su nombre de usuario, su email, la contraseña con su encriptación, su nombre, sus apellidos, la ruta de la foto de perfil y su área temática. También, se compone de un campo indicando si el usuario está activado y un campo que guarda la fecha del último inicio de sesión.
- Tabla **lugar**: se describe el nombre, una pequeña descripción, el id del lugar al que corresponde, la fecha de creación , la fecha de actualización , el creador y el actualizador,
- Tabla **fotografías**: se guarda el nombre de la foto, la ruta de la foto, indicaremos si esa foto está publicada, el punto de interés al que pertenece, la orden en la que ocupa, la fecha de creación, la fecha de la última actualización, el id de un creador y el de un actualizador y también el id de un área temática que pertenece.
- Tabla **usuario_lugar**: Es la tabla pivote que une tantos lugares como usuarios, es decir, un usuario podrá tener más de un lugar **(1, n)**, y un lugar podrá tener más de un usuario **(1, n)**.
- Tabla **punto de interés**: Es la tabla más importante, ya que va relacionada con muchas tablas, tendrá el qr, donde se generará un código qr a la hora de escanear, la distancia en la que se encuentra con el usuario, la latitud, la longitud, fechas de creación, de actualización, y el id de creador y el id de actualizador.
  Un punto de interés podrá tener muchos o ninguno **(0, n)**: vídeos, fotografías, visitas, áreas temáticas
  Un punto de interés sólo podrá pertenecer a un lugar **(1,1)**.
- Tabla **área temática**: solo tendrá un nombre y una descripción.

- Tabla **punto_area**: es la tabla pivote entre área temática y punto de interés, tendrá además un título, una gran descripción y un idioma.
  Por tanto un punto de interés puede tener muchas áreas temáticas **(1,n)** , y una área temática puede tener muchos puntos de intereses **(1,n)**.
- Tabla **visíta**: aqui se guardará información respecto al dispositivo del usuario, la hora, versión de la aplicación, sistema operativo, versión del sistema operativo, la latitud y la longitud donde se encuentra. También, guarda un punto de interés relacionado, ya que una visita puede tener ningún o varios puntos de interés **(0,n)**.
- Tabla **vídeos**: guardará un nombre el vídeo, una ruta del vídeo, indicaremos si ese vídeo está publicado, el orden en la que ocupa, el punto de interés y el área temática, ya que tanto para puntos de interés como áreas temáticas, un vídeo puede tener ninguno o muchos **(0,n)**.
- Tabla **vídeo_items**: se entiende que son las caracteristicas de vídeo. Guardará el id de vídeo relacionado, pues esas características de vídeo pueden tener en uno o varios vídeos **(1,n)**. También guarda la calidad, el formato, la orientación y el idioma.
- Tabla **roles**: proporcionada por la librería **laravel-permissions** para los roles.
- Tabla **permisos**: proporcionada por la librería **laravel-permissions** para los permisos.
- Tabla **model_has_role**: tabla pivote proporcionada por la librería **laravel-permissions** almacenando el id del usuario y el id del rol porque tiene relación **N:M**.
- Tabla **model_has_permissions**: tabla pivote proporcionada por la librería **laravel-permissions** almacenando el id del usuario y el id del permiso porque tiene relación **N:M**.
- Tabla **roles_has_permissions**: tabla pivote proporcionada por la librería **laravel-permissions** almacenando el id del rol y el id del permiso porque tiene relación **N:M**.

Migraciones

El primer paso que hemos realizado es la creación de la tablas gracias a las migraciones.

CreateUsersTable

La primera tabla que hemos creado se llama **users**, aunque ya estaba creada por defecto. Los campos son los siguientes: **id (auto_numerico)**, **login(string)**, **name(string)**, **surnames(string)**, **email(string)**, **profile(string)**, **password(string)**, **thematic_area_id(int, clave_foranea)**, **active(boolean)** y **last_login(datetime)**.

```php
<?php

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

class CreateUsersTable extends Migration
{
  public function up()
  {
    Schema::create('users', function (Blueprint $table) {
      $table->bigIncrements('id');
      $table->string('login', 100);
      $table->string('name', 200);
      $table->string('surnames', 200);
      $table->string('email', 100)->nullable();
      $table->string('profile', 100)->nullable();
      $table->string('password', 500)->nullable();
      $table->unsignedBigInteger('thematic_area_id')->nullable();
      $table->foreign('thematic_area_id')->on('thematic_areas')->references('id')->onDelete('cascade');
      $table->index(["thematic_area_id"], 'fk_user_area_thematicl_idx');
      $table->date('last_login')->nullable();
      $table->boolean('active')->default(true);
      $table->string('remember_token', 100)->nullable();;
      $table->timestamps();
      $table->softDeletes();
    });
  }

  public function down()
  {
    Schema::dropIfExists('users');
  }
}
```

CreatePlacesTable

La siguiente tabla que hemos creado se llama **places** cuyos campos son: **id (auto_numérico), name(string)**, **url(string), description(string)**, **place_id(int, clave_foranea)**, **date_create(database)**, **last_update(database)**, **creator(int, clave_foranea)** y **updater(int, clave_foranea)**.

```php
<?php

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

class CreatePlacesTable extends Migration
{
  public function up()
  {
    Schema::create('places', function (Blueprint $table) {
      $table->bigIncrements('id');
      $table->string('name', 45);
      $table->string('url', 45)->nullable();;
      $table->string('description', 45)->nullable();
      $table->unsignedBigInteger('place_id')->nullable();
      $table->foreign('place_id')->on('places')->references('id')->onDelete('cascade');
      $table->index(["place_id"], 'fk_place_place1_idx');
      $table->timestamp('date_create')->nullable();
      $table->timestamp('last_update')->nullable();
      $table->unsignedBigInteger('creator')->nullable();
      $table->foreign('creator')->on('users')->references('id')->onDelete('cascade');
      $table->index(["creator"], 'fk_place_user1_idx');
      $table->unsignedBigInteger('updater')->nullable();
      $table->foreign('updater')->on('users')->references('id')->onDelete('cascade');
      $table->index(["updater"], 'fk_place_user2_idx');
      $table->timestamps();
      $table->softDeletes();
    });
  }

  public function down()
  {
    Schema::dropIfExists('places');
  }
}
```

CreatePhotographiesTable

Otra de las tablas que hemos creado se denomina **photographies**. Los campos que se definen son los siguientes: **id(auto_numerico)**, **name(string)**, **url(string)**, **route(string)**, **point_of_interest_id(int, clave_foranea)**, **order(int)**, **published(boolean)**, **date_create(datetime)**, **last_update(datetime)**, **creator(int, clave_foranea)**, **updater(int, clave_foranea)** y **thematic_area_id(int, clave_foranea)**.

```php
<?php

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

class CreatePhotographiesTable extends Migration
{
  public function up()
  {
    Schema::create('photographies', function (Blueprint $table) {
      $table->bigIncrements('id');
      $table->string('name', 245);
      $table->string('url')->nullable();
      $table->string('route', 245)->nullable();
      $table->unsignedBigInteger('point_of_interest_id')->nullable();
      $table->foreign('point_of_interest_id')->on('point_of_interests')->references('id')->onDelete('cascade');
      $table->index(["point_of_interest_id"], 'fk_photography_points1_idx');
      $table->integer('order')->nullable();
      $table->boolean('published')->default(0);
      $table->timestamp('date_create')->nullable();
      $table->timestamp('last_update')->nullable();
      $table->unsignedBigInteger('creator')->nullable();
      $table->foreign('creator')->on('users')->references('id')->onDelete('cascade');
      $table->index(["creator"], 'fk_photography_user1_idx');
      $table->unsignedBigInteger('updater')->nullable();
      $table->foreign('updater')->on('users')->references('id')->onDelete('cascade');
      $table->index(["updater"], 'fk_photography_user2_idx');
      $table->unsignedBigInteger('thematic_area_id')->nullable();
      $table->foreign('thematic_area_id')->on('thematic_areas')->references('id');
      $table->index(["thematic_area_id"], 'fk_photography_thematic_area1_idx');
      $table->timestamps();
      $table->softDeletes();
    });
  }
}
```

CreatePointOfInterestsTable

Contamos también con la tabla **points_of_interests** cuyos campos son los siguientes: **id(auto_numerico)**, **qr(string)**, **url(string)**, **distance(int)**, **latitude(double)**, **longitude(double)**, **place_id(int, clave_foranea)**, **creation_date(database)**, **last_update_date(database)**, **creator(int, clave_foranea)** y **updater(int, clave_foranea)**.

```php
<?php

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

class CreatePointOfInterestsTable extends Migration
{
  public function up()
  {
    Schema::create('point_of_interests', function (Blueprint $table) {
      $table->bigIncrements('id');
      $table->string('qr',45)->nullable();
      $table->string('url',45)->nullable();
      $table->integer('distance')->nullable();
      $table->decimal('latitude',10,7)->nullable();
      $table->decimal('longitude',10,7)->nullable();
      $table->unsignedBigInteger('place_id')->nullable();
      $table->dateTime('creation_date');
      $table->datetime('last_update_date')->nullable();
      $table->unsignedBigInteger('creator');
      $table->unsignedBigInteger('updater')->nullable();
      $table->timestamps();
      $table->index('updater', 'fk_points_of_interest_user2_idx');
      $table->index('place_id', 'fk_points_of_interest_place_idx');
      $table->index('creator', 'fk_points_of_interest_user1_idx');
      $table->foreign('place_id','fk_points_of_interest_place_idx')->on('places')->references('id')->onDelete('cascade');
$table->foreign('creator','fk_points_of_interest_user1_idx')->on('users')->references('id')->onDelete('cascade');
      $table->foreign('updater','fk_points_of_interest_user2_idx')->on('users')->references('id')->onDelete('cascade');
      $table->softDeletes();
    });
  }

  public function down()
  {
    Schema::dropIfExists('point_of_interests');
  }
}
```

CreateVisitsTable

Otra de las tabla que vamos a crear se llama **visits**. Los campos que componen esta tabla son los siguientes: **id(auto_numerico)**, **hour(datetime)**, **deviceid(string)**, **url(string)**, **appversion(string)**, **useragent(string)**, **ssoo(string)**, **ssooversion(string)**, **latitude(double)**, **longitude(double)** y **point_of_interest(int, clave_foranea)**.

```php
<?php

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

class CreateVisitsTable extends Migration
{
  public function up()
  {
    Schema::create('visits', function (Blueprint $table) {
      $table->bigIncrements('id');
      $table->dateTime('hour')->nullable();
      $table->string('deviceid',85);
      $table->string('url', 85)->nullable();
      $table->string('appversion',45)->nullable();
      $table->string('useragent',95)->nullable();
      $table->string('ssoo',45)->nullable();
      $table->string('ssooversion',45)->nullable();
      $table->decimal('latitude',10,7)->nullable();
      $table->decimal('longitude',10,7)->nullable();
      $table->unsignedBigInteger('point_of_interest_id')->nullable();
      $table->timestamps();
      $table->index('point_of_interest_id','fk_visit_point_of_interest1_idx');
$table->foreign('point_of_interest_id','fk_visit_point_of_interest1_idx')->on('point_of_interests')->references('id')->onDelete('cascade');
      $table->softDeletes();
    });
  }

  public function down()
  {
    Schema::dropIfExists('visits');
  }
}
```

CreateThematicAreasTable

También hemos creado una tabla para los departamentos llamada **thematic_areas** con estos campos: **id(auto_numerico)**, **name(string)**, **url(string)** y **description(string)**.

```php
<?php

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

class CreateThematicAreasTable extends Migration
{
  public function up()
  {
    Schema::create('thematic_areas', function (Blueprint $table) {
      $table->bigIncrements('id');
      $table->string('name',45)->nullable();
      $table->string('url',45)->nullable();
      $table->string('description',245)->nullable();
      $table->timestamps();
      $table->softDeletes();
    });
  }

  public function down()
  {
    Schema::dropIfExists('thematic_areas');
  }
}
```

CreateVideosTable

La siguiente tabla para los vídeos se denomina **videos**. Los campos que representa esta tabla son los siguientes: **id(auto_numerico)**, **name(string)**, **url(string)**, **route(string)**, **point_of_interest_id(int, clave_foranea)**, **order(int)**, **published(boolean)**, **date_create(datetime)**, **last_update(datetime)**, **creator(int, clave_foranea)**, **updater(int, clave_foranea)** y **thematic_area_id(int, clave_foranea)**.

```php
<?php

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

class CreateVideosTable extends Migration
{
  public function up()
  {
    Schema::create('videos', function (Blueprint $table) {
      $table->bigIncrements('id');
      $table->string('name', 245);
      $table->string('url', 245)->nullable();
      $table->string('route', 245)->nullable();
      $table->unsignedBigInteger('point_of_interest_id')->nullable();
      $table->foreign('point_of_interest_id')->on('point_of_interests')->references('id')->onDelete('cascade');
      $table->index['point_of_interest_id'], 'fk_video_points_of_interest1_idx');
      $table->integer('order')->nullable();
      $table->boolean('published')->default(0);
      $table->timestamp('date_create')->nullable();
      $table->timestamp('last_update')->nullable();
      $table->unsignedBigInteger('creator')->nullable();
      $table->foreign('creator')->on('users')->references('id')->onDelete('cascade');
      $table->index['creator'], 'fk_video_user1_idx');
      $table->unsignedBigInteger('updater')->nullable();
      $table->foreign('updater')->on('users')->references('id')->onDelete('cascade');
      $table->index['updater'], 'fk_video_user2_idx');
      $table->unsignedBigInteger('thematic_area_id')->nullable();
      $table->foreign('thematic_area_id')->on('thematic_areas')->references('id');
      $table->index['thematic_area_id'], 'fk_video_thematic_area1_idx');
      $table->timestamps();
      $table->softDeletes();
    });
  }

}
```

CreateVideoItemsTable

Para las características de vídeo, hemos creado una tabla denominada **video_items**, que cuenta con los siguientes campos: **id(auto_numerico)**, **video_id(int, clave_foranea)**, **format(string)**, **quality(string)**, **orientation(string)** y **language(string)**.

```php
<?php

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

class CreateVideoItemsTable extends Migration
{
  public function up()
  {
    Schema::create('video_items', function (Blueprint $table) {
      $table->bigIncrements('id');
      $table->unsignedBigInteger('video_id');
      $table->foreign('video_id')->on('videos')->references('id')->onDelete('cascade');
      $table->index(['video_id'], 'fk_video_item_video1_idx');
      $table->string('url', 245)->nullable();
      $table->string('quality', 45)->nullable();
      $table->string('format', 45)->nullable();
      $table->string('orientation')->nullable();
      $table->string('language', 10)->nullable();
      $table->timestamps();
      $table->softDeletes();
    });
  }

  public function down()
  {
    Schema::dropIfExists('video_items');
  }
}
```

CreateUserPlaceTable

En el diseño de la base de datos, teníamos dos tablas pivote. La primera de ellas se llama **user_place**, donde tenemos el id de un usuario y el id de un lugar gracias a los campos: **user_id(int, clave_foranea)** y **place_id(int, clave_foranea)**.

```php
<?php

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

class CreateUserPlaceTable extends Migration
{
  public function up()
  {
    Schema::create('place_user', function (Blueprint $table) {
      $table->unsignedBigInteger('user_id');
      $table->foreign('user_id')->on('users')->references('id');
      $table->index(['user_id'], 'fk_place_has_user_user1_idx');
      $table->unsignedBigInteger('place_id');
      $table->foreign('place_id')->on('places')->references('id')->onDelete('cascade');
      $table->index(['place_id'], 'fk_place_has_user_place1_idx');
    });
  }

  public function down()
  {
    Schema::dropIfExists('place_user');
  }
}
```

CreatePointOfInterestThematicAreaTable

La segunda tabla pivote se llama **point_of_interest_thematic_area** donde almacenamos el id de punto de interés y el id de área temática. A esa fila, insertamos una serie de datos. Los campos son los siguientes: **thematic area id(int, clave foránea)**, **point_of_interest_id(int, clave foránea)**, **title(string)**, **description(string)** y **language(string)**.

```php
<?php

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

class CreatePointOfInterestThematicAreaTable extends Migration
{
  public function up()
  {
    Schema::create('point_of_interest_thematic_area', function (Blueprint $table) {
      $table->unsignedBigInteger('thematic_area_id')->nullable();
      $table->unsignedBigInteger('point_of_interest_id')->nullable();
      $table->string('title',245)->default('lorem ipsum');
      $table->string('description',245)->nullable();
      $table->string('language',10)->default('es');
      $table->index('point_of_interest_id','fk_thematic_area_has_point_of_interest_point_of_interest1_idx');
      $table->index('thematic_area_id','fk_thematic_area_id_has_point_of_interest_thematic_area1_idx');
$table->foreign('thematic_area_id','fk_thematic_area_id_has_point_of_interest_thematic_area1_idx')
        ->on('thematic_areas')
        ->references('id')->onDelete('cascade');
$table->foreign('point_of_interest_id','fk_thematic_area_has_point_of_interest_point_of_interest1_idx')
        ->on('point_of_interests')
        ->references('id')->onDelete('cascade');
    });
  }

  public function down()
  {
    Schema::dropIfExists('point_of_interest_thematic_area');
  }
}
```

ANOTACIÓN: teniendo instalada la librería laravel-permissions, tendríamos la migración **CreatePermissionTables** para la creación de las tablas: roles, permissions, model_has_roles, model_has_permissions y roles_has_permissions.

Seeders

Con los seeders, hemos generado una serie de datos que serán registrados en la base de datos. La mayoría de los seeders, creará datos falsos a través de las factories. Los seeders los hemos creado ejecutando el comando **php artisan make:seed [nombre_seeder]**. Estos son los seeders que hemos desarrollado:

DatabaseSeeder

En primer lugar, hemos definido el orden a la hora de cargar los seeders.

```php
<?php

use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
  public function run()
  {
    $this->call(ThematicAreasTableSeeder::class);
    $this->call(UsersTableSeeder::class);
    $this->call(PlacesTableSeeder::class);
    $this->call(PointOfInterestsTableSeeder::class);
    $this->call(PhotographiesTableSeeder::class);
    $this->call(VisitsTableSeeder::class);
    $this->call(VideosTableSeeder::class);
    $this->call(Video_itemsTableSeeder::class);
  }
}
```

UsersTableSeeder

Antes de generar usuarios falsos en el seeder, hemos realizado lo siguiente. Primero, cuando recarguemos los seeders, borramos una carpeta en Storage llamada **users** donde guardaremos una foto de perfil. Después, la volveremos a crear pero asignando permisos de escritura y lectura. Después, vamos a registrar cuatro roles: **Super Administrador**, **Administrador**, **Profesor** y **Alumno**.

```php
<?php

use App\User;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\Storage;
use Spatie\Permission\Models\Permission;
use Spatie\Permission\Models\Role;

class UsersTableSeeder extends Seeder
{
  public function run()
  {
    Storage::disk('public')->deleteDirectory('users');
    Storage::makeDirectory('/public/users');
    chmod(base_path().'/storage/app/public/users', 0777);

    $superAdmin = Role::create(['name' => 'Super Administrador', 'display_name' => 'Super Admin']);
    $admin = Role::create(['name' => 'Administrador','display_name'=>'Admin']);
    $teacher = Role::create(['name' => 'Profesor','display_name'=>'Teacher']);
    $student = Role::create(['name' => 'Alumno','display_name'=>'Student']);
```

Además, vamos a crear una serie de permisos donde, más adelante, serán asignados a un rol en concreto o a un usuario. Los permisos van a ser los siguientes:

- Roles: **Crear roles**, **Editar roles** y **Borrar roles**.
- Usuarios: **Ver usuarios**, **Crear usuarios**, **Editar usuarios** y **Borrar usuarios**.
- Lugares: **Ver lugares**, **Crear lugares**, **Editar lugares** y **Borrar lugares**.
- Fotografias: **Ver fotografias**, **Crear fotografias**, **Editar fotografias** y **Borrar fotografias**.

- vídeos: **Ver vídeos**, **Crear vídeos**, **Editar vídeos** y **Borrar vídeos**.
- Usuarios: **Ver usuarios**, **Crear usuarios**, **Editar usuarios** y **Borrar usuarios**.
- Puntos de interés: **Ver puntos de interes**, **Crear puntos de interes**, **Editar puntos de interes** y **Borrar puntos de interes**.
- Áreas temáticas: **Ver areas tematicas**, **Crear areas tematicas**, **Editar areas tematicas** y **Borrar areas tematicas**.
- Y también, una serie de permisos adicionales: **Ver ranking de puntos de interés**, **Ver gráfico de vistas**, **Ver gráfico de creaciones de puntos de interés**, **Ver contador de nuevos registros** y **Ver contador de fotos y vídeos registrados**.

```
$createRolePermission = Permission::create(['name' => 'Crear roles']);
$updateRolePermission = Permission::create(['name' => 'Editar roles']);
$deleteRolePermission = Permission::create(['name' => 'Borrar roles']);

$viewUserPermission = Permission::create(['name' => 'Ver usuarios']);
$createUserPermission = Permission::create(['name' => 'Crear usuarios']);
$updateUserPermission = Permission::create(['name' => 'Editar usuarios']);
$deleteUserPermission = Permission::create(['name' => 'Borrar usuarios']);

$viewPhotographiesPermission = Permission::create(['name' => 'Ver fotografias']);
$createPhotographiesPermission = Permission::create(['name' => 'Crear fotografias']);
$updatePhotographiesPermission = Permission::create(['name' => 'Editar fotografias']);
$deletePhotographiesPermission = Permission::create(['name' => 'Borrar fotografias']);

$viewVideosPermission = Permission::create(['name' => 'Ver videos']);
$createVideosPermission = Permission::create(['name' => 'Crear videos']);
$updateVideosPermission = Permission::create(['name' => 'Editar videos']);
$deleteVideosPermission = Permission::create(['name' => 'Borrar videos']);

$viewPointsOfInterestPermission = Permission::create(['name' => 'Ver puntos de interes']);
$createPointsOfInterestPermission = Permission::create(['name' => 'Crear puntos de interes']);
$updatePointsOfInterestPermission = Permission::create(['name' => 'Editar puntos de interes']);
$deletePointsOfInterestPermission = Permission::create(['name' => 'Borrar puntos de interes']);

$viewPlacesPermission = Permission::create(['name' => 'Ver lugares']);
$createPlacesPermission = Permission::create(['name' => 'Crear lugares']);
$updatePlacesPermission = Permission::create(['name' => 'Editar lugares']);
$deletePlacesPermission = Permission::create(['name' => 'Borrar lugares']);

$viewThematicAreasPermission = Permission::create(['name' => 'Ver areas tematicas']);
$createThematicAreasPermission = Permission::create(['name' => 'Crear areas tematicas']);
$updateThematicAreasPermission = Permission::create(['name' => 'Editar areas tematicas']);
$deleteThematicAreasPermission = Permission::create(['name' => 'Borrar areas tematicas']);

$charRankingtPermission=Permission::create(['name' => 'Ver ranking de puntos de intereses']);
$chartVisitsPermission=Permission::create(['name' => 'Ver gráfico de visitas']);
$chartPointOfInterestPermission=Permission::create(['name' => 'Ver gráfico de creaciones de puntos de intereses']);
$chartNewsRegistrationPermission=Permission::create(['name' => 'Ver contador de nuevos registros']);
$chartCountPhotosAndVideosPermission=Permission::create(['name' => 'Ver contador de fotos y videos registrados']);
```

Después, asignamos estos permisos a los roles. Por lo que quedaría de la siguiente manera:

- Un alumno no solo puede crear y editar tanto las fotos como los vídeos sino puede ver un contador de fotos y vídeos, y además, puede ver el ranking de los puntos de interés más visitados.
- Un profesor puede ver, editar y borrar tanto las fotos como vídeos porque son los que se van a encargar de decidir si quieren publicar o no. Solo podrá ver las fotos o vídeos asociado a su rama temática.
- Además, el profesor puede ver, crear y editar los lugares. Lo mismo ocurrirá con los puntos de interés y las áreas temáticas. También, puede ver el contador de fotos y vídeos registrados y el ranking con los puntos de interés más visitados.

```
$student->givePermissionTo([$createPhotographiesPermission, $updatePhotographiesPermission,
  $createVideosPermission, $updateVideosPermission,
  $chartCountPhotosAndVideosPermission, $charRankingtPermission
]);
$teacher->givePermissionTo([$viewPhotographiesPermission,
  $updatePhotographiesPermission, $deletePhotographiesPermission,
  $viewVideosPermission, $updateVideosPermission, $deleteVideosPermission,
  $viewPointsOfInterestPermission, $createPointsOfInterestPermission, $updatePointsOfInterestPermission,
  $viewPlacesPermission, $createPlacesPermission, $updatePlacesPermission,
  $viewThematicAreasPermission, $createThematicAreasPermission, $updateThematicAreasPermission,
  $chartCountPhotosAndVideosPermission, $charRankingtPermission
]);
```

A continuación, creamos nuestros usuarios administradores con nuestros datos y la contraseña de acceso al panel de control. También añadimos un usuario concreto con el rol de alumno. Por cierto, uno de estos administradores será el superadministrador.

```
$user = new User;
$user->login = 'pepe123';
$user->name = 'Pepe';
$user->surnames = 'Martinez Lopez';
$user->password = 'hola';
$user->email = 'pepe@gmail.com';
$user->save();

$user->assignRole($student);

$user = new User;
$user->login = 'fran9614';
$user->name = 'Fran';
$user->surnames = 'Arce Codina';
$user->email = 'franarcecodina96@gmail.com';
$user->password = 'daw_2019';
$user->save();

$user->assignRole($admin);

$user = new User;
$user->login = 'jorgicoor1998@gmail.com';
$user->login = 'jorgicoor1998';
$user->name = 'Jorge';
$user->surnames = 'Orenes Rubio';
$user->email = 'jorgicoor1998@gmail.com';
$user->password = 'pokemon12';
$user->save();

$user->assignRole($admin);

$user = new User;
$user->login = 'fulgen_daw123';
$user->name = 'Fulgencio';
$user->surnames = 'Valera Alonso';
$user->password = '654321';
$user->email = 'fulgencio.valera@gmail.com';
$user->save();
$user->assignRole($superAdmin);
$user->assignRole($admin);
```

Completando el seeder, generamos 10 usuarios falsos con el rol profesor.

```
$users = factory(User::class,10)->make();

    $users->each(function($u) use($teacher) {
      $u->save();
      $u->assignRole($teacher);
    });
  }
}
```

PlacesTableSeeder

Vamos a generar 10 lugares con datos falsos. A medida que vaya generando esos datos falsos, se irá registrando la tabla pivote **user_place**.

```php
<?php

use App\Place;
use App\User;
use Illuminate\Database\Seeder;
use Illuminate\Support\Arr;
use Illuminate\Support\Str;

class PlacesTableSeeder extends Seeder
{
  public function run()
  {
    $places = factory(Place::class,10)->create();
    $places->each(function($p) {
      $p->url = Str::slug($p->name);
      $p->save();
      $user = User::all()->pluck('id')->toArray();
      $p->users()->attach(Arr::random($user, 2));
    });
  }
}
```

PhotographiesTableSeeder

Vamos a generar 30 lugares con datos falsos. Además, queremos que se borre la carpeta de Storage de las fotos a la hora de recargar este seeder.

```php
<?php

use App\Photography;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Str;

class PhotographiesTableSeeder extends Seeder
{
  public function run()
  {
    Storage::disk('public')->deleteDirectory('photos');
    $photography = factory(Photography::class, 30)->make();
    $photography->each(function($p) {
      $p->url = Str::slug($p->name);
      $p->save();
    });
  }
}
```

PointsOfInterestTableSeeder

Vamos a generar 20 puntos de interés con datos falsos. Como va a ser uno de los campos de la tabla pivote, obtendremos las áreas temáticas en array con el fin de actualizar la tabla pivote.

```php
<?php

use App\PointOfInterest;
use App\ThematicArea;
use Illuminate\Database\Seeder;
use Illuminate\Support\Arr;
use Illuminate\Support\Str;

class PointOfInterestsTableSeeder extends Seeder
{
  public function run()
  {
    $pointsInterest = factory(PointOfInterest::class, 20)->make();
    $pointsInterest->each(function($p) {
      $faker = Faker\Factory::create();
      $p->url = Str::slug($p->qr);
      $p->save();
      $thematicAreas= ThematicArea::all()->pluck('id')->toArray();
      $p->thematicAreas()->attach(Arr::random($thematicAreas, 2),
        [
          'title' =>  $faker->sentence,
          'description'   =>  $faker->text,
          'language'  =>$faker->languageCode
        ]);
    });
  }
}
```

VisitsTableSeeder

Vamos a generar 30 visitas con datos falsos.

```php
<?php


use Illuminate\Database\Seeder;

use App\Visit;

use Illuminate\Support\Str;


class VisitsTableSeeder extends Seeder

{

  public function run()

  {

    $visits = factory(Visit::class, 30)->make();

    $visits->each(function($v) {

      $v->url = Str::slug($v->deviceid);

      $v->save();

    });

  }

}
```

ThematicAreasTableSeeder

A continuación, vamos a crear 10 áreas temáticas falsas.

```php
<?php

use App\ThematicArea;
use Illuminate\Database\Seeder;
use Illuminate\Support\Str;

class ThematicAreasTableSeeder extends Seeder
{
  public function run()
  {
    $thematic_areas = factory(ThematicArea::class,10)->make();
    $thematic_areas->each(function($v) {
      $v->url = Str::slug($v->name);
      $v->save();
    });
  }
}
```

VideosTableSeeder

También vamos a generar 10 vídeos falsos. Además, tendremos una carpeta Storage para los vídeos ya que se borrara cada vez que se recargue este seeder.

```php
<?php

use App\Video;
use Illuminate\Database\Seeder;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Str;

class VideosTableSeeder extends Seeder
{
  public function run()
  {
    Storage::disk('public')->deleteDirectory('videos');

    $video = factory(Video::class,10)->make();
    $video->each(function($v) {
      $v->url = Str::slug($v->route);
      $v->save();
    });
  }
}
```

Video_itemTableSeeder

Para terminar, vamos a generar para los vídeos falsos 30 características de video.

```php
<?php

use App\VideoItem;
use Illuminate\Database\Seeder;
use Illuminate\Support\Str;

class Video_itemsTableSeeder extends Seeder
{
  public function run()
  {
    $video_item = factory(VideoItem::class, 30)->make();
    $video_item->each(function($v) {
      $v->url = Str::slug($v->video->route);
      $v->save();
    });
  }
}
```

Factories

Ejecutando el comando **php artisan make:factory [nombre_factory]** hemos creado una serie de factories necesarios:

UserFactory

Desarrollo basado en la generación de datos falsos para los usuarios.

```php
<?php

use App\ThematicArea;
use App\User;
use Faker\Generator as Faker;

$factory->define(User::class, function (Faker $faker) {
  return [
    'login' => $faker->userName,
    'name' => $faker->firstName,
    'surnames' => $faker->lastName . ' ' . $faker->lastName,
    'email' => $faker->unique()->safeEmail,
    'password' => '123456',
    'thematic_area_id' => $faker->randomElement(ThematicArea::all()->pluck('id')->toArray()),
    'active' => $faker->boolean()
  ];
});
```

PlaceFactory

Desarrollo basado en la generación de datos falsos para los lugares.

```php
<?php

use App\Place;
use App\User;
use Faker\Generator as Faker;

$factory->define(\App\Place::class, function (Faker $faker) {
  $user = User::all()->pluck('id')->toArray();
  return [
    'name'=> $faker->city,
    'description'=> $faker->word,
    'place_id'=> $faker->randomElement(Place::all()->pluck('id')->toArray()),
    'creator' => $faker->randomElement($user),
    'updater' => $faker->randomElement($user),
    'date_create'=> $faker->dateTimeThisMonth,
    'last_update' => $faker->dateTimeThisMonth,
  ];
});
```

PhotographyFactory

Desarrollo basado en la generación de datos falsos para las fotografías.

```php
<?php

use App\Photography;
use App\PointOfInterest;
use App\ThematicArea;
use App\User;
use Faker\Generator as Faker;

$factory->define(Photography::class, function (Faker $faker) {
  return [
    'name'  => $faker->sentence,
    'point_of_interest_id' => $faker->randomElement(PointOfInterest::all()->pluck('id')->toArray()),
    'order' => $faker->randomDigit,
    'published' => $faker->boolean,
    'creator' => $faker->randomElement(User::all()->pluck('id')->toArray()),
    'updater' => $faker->randomElement(User::all()->pluck('id')->toArray()),
    'thematic_area_id' => $faker->randomElement(ThematicArea::all()->pluck('id')->toArray()),
    'date_create' => $faker->dateTimeThisMonth,
    'last_update' => $faker->dateTimeThisMonth,
  ];
});
```

PointOfInterestFactory

Desarrollo basado en la generación de datos falsos para los puntos de interés.

```php
<?php

use App\Place;
use App\PointOfInterest;
use App\User;
use Faker\Generator as Faker;
use Illuminate\Support\Str;

$factory->define(PointOfInterest::class, function (Faker $faker) {
  return [
    'qr' => Str::random(35),
    'distance' => $faker->randomNumber(2),
    'latitude' => $faker->latitude($min = 3, $max = 20),
    'longitude' => $faker->longitude($min = 3, $max = 20),
    'creator'=> $faker->randomElement(User::all()->pluck('id')->toArray()),
    'updater' => $faker->randomElement(User::all()->pluck('id')->toArray()),
    'place_id'=>$faker->randomElement(Place::all()->pluck('id')->toArray()),
    'creation_date' => $faker->dateTimeThisMonth,
    'last_update_date' => $faker->dateTimeThisMonth,
  ];
});
```

VisitFactory

Desarrollo basado en la generación de datos falsos para las visitas.

```php
<?php

use App\PointOfInterest;
use App\Visit;
use Faker\Generator as Faker;

$factory->define(Visit::class, function (Faker $faker) {
  return [
    'hour' => $faker->dateTime(),
    'deviceid' => $faker->uuid,
    'appversion' => $faker->numberBetween(1, 10),
    'useragent' => $faker->word,
    'ssoo' => $faker->word,
    'ssooversion' => $faker->word,
    'latitude' => $faker->latitude,
    'longitude' => $faker->longitude,
    'point_of_interest_id' => $faker->randomElement(PointOfInterest::all()->pluck('id')->toArray())
  ];
});
```

ThematicAreaFactory

Desarrollo basado en la generación de datos falsos para las áreas temáticas.

```php
<?php

use Faker\Generator as Faker;

$factory->define(\App\ThematicArea::class, function (Faker $faker) {
  return [
    'name'=> $faker->word,
    'description'=> $faker->sentence(2)
  ];
});
```

VideoFactory

Desarrollo basado en la generación de datos falsos para los vídeos.

```php
<?php

use App\PointOfInterest;
use App\ThematicArea;
use App\User;
use App\Video;
use Faker\Generator as Faker;

$factory->define(Video::class, function (Faker $faker) {
  return [
    'name'  =>  $faker->sentence,
    'order'=> $faker->randomDigit,
    'published' => $faker->boolean,
    'thematic_area_id' => $faker->randomElement(ThematicArea::all()->pluck('id')->toArray()),
    'creator' => $faker->randomElement(User::all()->pluck('id')->toArray()),
    'updater' => $faker->randomElement(User::all()->pluck('id')->toArray()),
    'point_of_interest_id' => $faker->randomElement(PointOfInterest::all()->pluck('id')->toArray()),
    'date_create' => $faker->dateTimeThisMonth,
    'last_update' => $faker->dateTimeThisMonth,
  ];
});
```

VideoItemFactory

Desarrollo basado en la generación de datos falsos para las características de vídeo.

```php
<?php

use App\Video;
use App\VideoItem;
use Faker\Generator as Faker;

$factory->define(VideoItem::class, function (Faker $faker) {
  $qualities = array('360p', '480p', '720p', '1080p', '4K');
  $formats = array('avi', 'mp4', 'ogg');
  $orientations = array('horizontal', 'vertical');

  return [
    'video_id' => $faker->randomElement(Video::all()->pluck('id')->toArray()),
    'quality' => $faker->randomElement($qualities),
    'format' => $faker->randomElement($formats),
    'orientation' => $faker->randomElement($orientations),
    'language' => $faker->languageCode
  ];
]);
```

Rutas

routes/web.php

Definimos un grupo de rutas para el acceso a todo el panel de administración. Algunas rutas proporcionan seguridad a través de los middlewares.

```php
<?php
Route::get('/', 'Auth\LoginController@showLoginForm')->name('login');
Route::group([
  'prefix'=>'admin',
  'namespace'=>'Admin',
  'middleware'=>'auth',
], function() {
  Route::get('home', 'AdminController@index')->name('admin.dashboard');

  Route::resource('users', 'UsersController', ['as' => 'admin'], ['middleware' => 'Ver usuarios', 'Crear usuarios', 'Editar usuarios']);

  Route::get('users/{user}/photo','UsersController@editphoto')->name('admin.users.photo');
  Route::post('users/photo/update','UsersController@updatePhoto')->name('admin.users.photo.update');

  Route::resource('places', 'PlacesController', ['except' => ['create'], 'as' => 'admin']);
  Route::resource('photographies', 'PhotographiesController', ['except' => ['create'], 'as' => 'admin']);
  Route::resource('pointsofinterest', 'PointsOfInterestController', ['except' => ['create'], 'as' => 'admin']);
  Route::resource('visits', 'VisitsController', ['except' => ['create'], 'as' => 'admin']);
  Route::resource('thematicareas', 'ThematicAreasController', ['except' => ['create'], 'as' => 'admin']);
  Route::resource('videos', 'VideosController', ['except' => ['create'], 'as' => 'admin']);
  Route::resource('videoitems', 'VideoItemsController', ['except' => ['index, create, store, show, destroy'], 'as' => 'admin']);

  Route::resource('roles', 'RolesController', ['except' => 'show', 'as' => 'admin'], ['middleware' => 'Crear roles', 'Editar roles']);

  Route::middleware('role:Administrador')
    ->put('users/{user}/roles', 'UsersRolesController@update')
    ->name('admin.users.roles.update');
  Route::middleware('role:Administrador')
    ->put('users/{user}/permissions', 'UsersPermissionsController@update')
    ->name('admin.users.permissions.update');
});
```

Además, hemos definido las rutas para el login, el logout y el reseteo de contraseña. En el reseteo de contraseña, las rutas son de acceso al correo y a la nueva contraseña.

```php
Route::get('login', 'Auth\LoginController@showLoginForm')->name('login');

Route::post('login', 'Auth\LoginController@login');
Route::post('logout', 'Auth\LoginController@logout')->name('logout');

Route::get('password/email', 'Auth\ForgotPasswordController@showLinkRequestForm')->name('password.email');
Route::post('password/email', 'Auth\ForgotPasswordController@sendResetLinkEmail');

Route::post('password/reset', 'Auth\ResetPasswordController@reset') ;
Route::get('password/reset', 'Auth\ForgotPasswordController@showLinkRequestForm')->name('password.request');
Route::get('password/reset/{token}', 'Auth\ResetPasswordController@showResetForm')->name('password.reset');
Route::post('password/reset', 'Auth\ResetPasswordController@reset')->name('password.update');
```

Modelos

Una vez creadas todas las tablas con sus datos falsos (factories y los seeders), tenemos creados los modelos que representan las tablas con el fin de obtener, listar, crear, actualizar y borrar los datos de las tablas. También, tendremos definidos las relaciones de entidades, si las hay, los mutators, los disparadores y los scopes (las consultas). Además, aplicamos a cada modelo el uso de borrado blando o nativo **(SoftDeletes)**. Los modelos que hemos desarrollado son los siguientes:

Clase User

Definimos estas siguientes clases para el modelo **User**.

```php
<?php

namespace App;

use Carbon\Carbon;
use DateTime;
use Illuminate\Auth\Events\Login;
use Illuminate\Database\Eloquent\SoftDeletes;
use Illuminate\Foundation\Auth\User as Authenticatable;
use Illuminate\Notifications\Notifiable;
use Illuminate\Support\Facades\Request;
use Illuminate\Support\Facades\Storage;
use Spatie\Permission\Traits\HasRoles;
use App\Notifications\MyResetPassword;
```

Recordemos que este modelo se creó automáticamente cuando se creó el proyecto. Usamos las clases de la librería **laravel-permissions** para poder aplicar los roles y permisos a los usuarios. Indicamos los campos que hagan una asignación masiva: **login**, **name**, **surnames**, **email**, **password**, **profile**, **thematic_area_id** y **active**.

```php
class User extends Authenticatable
{

  use Notifiable, hasRoles, SoftDeletes;

  protected $fillable = [ 'login', 'name', 'surnames', 'email', 'password', 'profile', 'thematic_area_id', 'active'];

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

  protected $casts = ['email_verified_at' => 'datetime'];

  public static function create(array $attributes = [])
  {
    $user = static::query()->create($attributes);
    return $user;
  }
```

Después, definimos las relaciones con otros modelos:

- **Lugares →** uno o varios usuarios puede tener muchos lugares.
- **Fotografías →** un usuario puede tener muchas fotografías.
- **Punto de interés →** un usuario puede tener muchos puntos de interés.

- **Área temática →** un usuario puede o no estar en un área temática.
- **vídeos →** un usuario puede tener muchos vídeos.

```php
public function places()
{
    return $this->belongsToMany(Place::class);
}

public function photographies()
{
    return $this->hasMany(Photography::class, 'creator');
}

public function points_of_interest()
{
    return $this->hasMany(PointOfInterest::class, 'creator');
}

public function thematic_area()
{
    return $this->belongsTo(ThematicArea::class);
}

public function videos()
{
    return $this->hasMany(Video::class, 'creator');
}
```

Siguiendo el modelo, hemos desarrollado:

- Disparadores a la hora de:
    - **Actualizar →** actualiza el área temática con su valor o como null.
    - **Borrar →** borra una foto de perfil desde la carpeta Storage, y sus roles y permisos asignados.
- Un mutator que encripta la contraseña cada vez que se actualice.
- Un método indicando si un usuario puede ver todos los usuarios o solo el suyo en el listado de usuarios.

```php
public static function boot()
{
    parent::boot();

    static::updating(function ($user) {
        Request::has('thematic_area_id') ? $user->thematic_area_id : null;
    });

    static::deleting(function ($user) {
        Storage::disk('public')->delete($user->profile);
        $user->roles()->detach();
        $user->permissions()->detach();
        $user->places()->detach();
    });
}

public function setPasswordAttribute($password)
{
    $this->attributes['password'] = bcrypt($password);
}

public function scopeAllowed($query)
{
    if (auth()->user()->can('view', $this)) {
        return $query;
    } else {
        return $query->where('id', auth()->id());
    }
}
```

Hemos desarrollado una serie de métodos interesantes a la hora de iniciar sesión. Esta parte corresponde al apartado denominado **Caducidad del usuario**:

- Guardamos fecha del último inicio de sesión.
- Consultamos si el usuario lleva más de tres meses sin loguearse. Si es cierto, pasaría a inactivo.

También, hemos desarrollado un método que envía una notificación de correo a la hora de resetear la contraseña del login.

```php
public static function registerLastLogin(Login $event)
{
    $event->user->last_login = new DateTime();
    $event->user->save();
}

public static function InactiveUsers()
{
    $date = Carbon::now();
    return User::query()->where([
        ['last_login', '<', $date->subMonth(3)],
        ['active', '=', true]
    ]);
}

public function sendPasswordResetNotification($token)
{
    $this->notify(new MyResetPassword($token));
}
```

Por último, implementamos un método que devuelve como entero el número de usuarios registrados durante las últimas 24 horas.

```php
public static function countNewUsers()
{
    return  (int)count(User::whereDate('created_at', Carbon::today())->get());
}
}
```

Clase Place

Para la clase **Place**, hemos definido las clases que se van a utilizar.

```php
<?php

namespace App;

use Carbon\Carbon;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;
use Illuminate\Support\Str;
```

A continuación, definimos aquellos campos que se les aplicaran una asignación masiva: **name**, **url**, **description**, **place_id**, **date_create**, **last_update**, **creator** y **updater**. Además, definimos los campos de tipo fecha: **date_create** y **last_update**.

Y las relaciones con otros modelos:

- **Usuarios →** uno o varios lugares puede ser creado por uno o varios usuarios.
- **Lugares →** un lugar podría tener un mismo lugar o ninguno.
- **Puntos de interés →** un lugar puede tener varios puntos de interés.

```php
class Place extends Model
{
  use SoftDeletes;

  protected $fillable = ['name', 'url', 'description', 'place_id', 'date_create', 'last_update', 'creator', 'updater'];
  protected $dates = ['date_create', 'last_update'];

  public function users()
  {
    return $this->belongsToMany(User::class);
  }

  public function place()
  {
    return $this->belongsTo(Place::class, 'place_id');
  }

  public function userCreator()
  {
    return $this->belongsTo(User::class, 'creator');
  }

  public function userUpdater()
  {
    return $this->belongsTo(User::class, 'updater');
  }

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

Continuando con el desarrollo del modelo, hemos desarrollado lo siguiente:

- Disparadores a la hora de:
  - **Crear →** registra una nueva fila en la tabla usuario_lugar.
  - **Actualizar →** actualiza la fecha de edición y la actualización de la tabla pivote usuario_lugar.
  - **Borrar →** el lugar que guarda en la tabla puntos de interés, pasará a **null**.
- Método estático **create()** donde registramos el id de usuario quien lo haya creado y la fecha de creación. Además, registramos un slug o url para el lugar.

```php
  public static function boot()
  {
    parent::boot();

    static::created(function ($place){
      $place->users()->attach($place->creator);
    });

    static::updating(function($place) {
      $place->last_update = Carbon::now();
      $place->syncusers($place->updater);
    });

    static::deleting(function($place) {
      $place->users()->detach();
      $place->pointsOfInterests()->each(function($p) {
        $p->place_id = null;
        $p->save();
      });
    });
  }

  public static function create(array $attributes = [])
  {
    $attributes['date_create'] = Carbon::now();
    $attributes['creator'] = auth()->user()->id;
    $place = static::query()->create($attributes);
    $place->generateSlug();
    return $place;
  }
```

Definimos un método que permite añadir o actualizar las filas de la tabla pivote **user_place**, comprobando, en un método, si ya existían esos registros para no tener filas repetidas.

```
public function syncusers($user)
{
  if($this->existsPlaceUserId($user)) {
     return $this->users()->sync($user);
  }
  return $this->users()->attach($user);
}

public function existsPlaceUserId($id)
{
  return $this->users()->where('place_id', '=', $this->id)->where('user_id', '=', $id)->exists();
}
```

También, hemos desarrollado un método que registra un slug o url mientras creamos un lugar, comprobando que no sea la misma url.
Para terminar, una consulta que compruebe si un usuario tiene permiso para ver todos los lugares. Si no tiene permiso, tendría que ver sus propios lugares creados siempre y cuando tenga el rol profesor.

```
  public function generateSlug()
  {
    $url = Str::slug($this->name);

    if(static::whereUrl($url)->exists()) {
       $url .= '--' . static::where('url', 'like', $url . '-%')->count();
    }

    $this->url = $url;
    $this->save();
  }

  public function getRouteKeyName()
  {
     return 'url';
  }

  public function scopeAllowed($query)
  {
     if(auth()->user()->can('view', $this)) {
        return $query;
     }else{
        if(auth()->user()->hasRole('Profesor')){
           return $query->where('creator', auth()->id())->orWhere('updater', auth()->id());
        }
        abort(403);
     }
  }
}
```

Clase Photography

El siguiente modelo a desarrollar es la clase Photography. En primer lugar, definiremos las clases que hemos usado en este modelo:

```php
<?php

namespace App;

use Carbon\Carbon;
use Illuminate\Database\Eloquent\SoftDeletes;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Support\Facades\Request;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Str;
```

Las columnas definidas para la asignación masiva son las siguientes: **name**, **url**, **route**, **point_of_interest_id**, **order**, **published**, **date_create**, **last_update**, **creator**, **updater**, **thematic_area_id**. Los campos cuyo tipo es 'database' son: **date_create** y **last_update**.
A continuación, tenemos las siguientes relaciones:

- **Punto de interés →** una fotografía ha de tener un punto de interés.
- **Área temática →** una fotografía va a tener asignado un área temática.
- **Usuarios →** una fotografía tendrá un usuario quién lo creó y un usuario quien lo actualizo.

```php
class Photography extends Model
{
  use SoftDeletes;

  protected $fillable = ['name', 'url', 'route', 'point_of_interest_id', 'order', 'published', 'date_create', 'last_update',
'creator', 'updater', 'thematic_area_id'];
  protected $dates = ['date_create', 'last_update'];

  public function point_of_interest()
  {
    return $this->belongsTo(PointOfInterest::class);
  }

  public function thematic_area()
  {
    return $this->belongsTo(ThematicArea::class);
  }

  public function userCreator()
  {
    return $this->belongsTo(User::class, 'creator');
  }

  public function userUpdater()
  {
    return $this->belongsTo(User::class, 'updater');
  }
```

Siguiendo el modelo, hemos desarrollado lo siguiente:

- El método estático **create()**, donde cada vez que registremos una nueva foto, guardaremos el id de usuario creado con su fecha de creación y una url para una foto.
- Disparadores a la hora de:
    - **Actualizar →** actualiza el id de usuario editado, la fecha de edición y, si hay foto, lo vamos a guardar en una carpeta Storage llamada **photos**.
    - **Borrar →** borramos la foto que estaba guardada en la carpeta **photos** de Storage.

```php
public static function create(array $attributes = [])
{
    $attributes['creator'] = auth()->user()->id;
    $attributes['date_create'] = Carbon::now();
    $photography = static::query()->create($attributes);
    $photography->generateSlug();
    return $photography;
}

public static function boot()
{
    parent::boot();

    static::updating(function($photography) {
        if(Request::has('route')) {
            $file_name = $photography->video . '.png';
            $photography->route = Request::file('route')->storeAs('public/photos', $file_name);
        }
        $photography->updater = auth()->user()->id;
        $photography->last_update = Carbon::now();
    });

    static::deleting(function($photography){
        Storage::disk('public')->delete($photography->route);
    });
}
```

A continuación, tenemos desarrollado lo siguiente:

- Un método donde registramos el slug cuando insertamos una nueva foto, comprobando que no sea la misma url.
- Recibiremos como parámetro de acceso el propio slug.
- Una doble consulta:
    - La primera es si el usuario puede ver todas las fotos. En caso contrario, solo podrá ver sus fotos.
    - La segunda es que si el usuario tiene el rol profesor, podría ver las fotos dentro de su área temática.

```php
public function generateSlug()
{
    $url = Str::slug($this->name);
    if(static::whereUrl($url)->exists()) {
        $url .= '--' . static::where('url', 'like', $url . '-%')->count();
    }
    $this->url = $url;
    $this->save();
}

public function getRouteKeyName()
{
    return 'url';
}

public function scopeAllowed($query)
{
    if(auth()->user()->can('view', $this)) {
        if (auth()->user()->hasRole('Profesor')){
            return $query->where('thematic_area_id', auth()->user()->thematic_area_id);
        }
        return $query;
    }else{
        return $query->where('creator', auth()->id())->orWhere('updater', auth()->id());
    }
}
```

Completando el modelo, implementamos un método que devuelve, como entero, el número de fotos creados durante las últimas 24 horas.

```
  public static function countNewPhotos()
  {
    return (int)count(Photography::whereDate('created_at', Carbon::today())->get());
  }
}
```

Clase PointOfInterest

Continuamos con la clase **PointOfInterest**. Hemos definido, para empezar, las clases que se han utilizado:

```
<?php

namespace App;

use Carbon\Carbon;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;
use Illuminate\Support\Str;
```

Las columnas que se les realizará una asignación masiva son las siguientes: **qr**, **distance**, **longitude**, **creator**, **updater**, **latitude**, **creation_date** y **last_update_date**. Los campos de tipo **database** son: **creation_date** y **last_update_date**.

```
class PointOfInterest extends Model
{
  use SoftDeletes;

  protected $fillable = ['qr','distance','longitude', 'creator', 'updater', 'place_id','latitude','creation_date', 'last_update_date'];
  protected $dates = ['created_at', 'updated_at',  'creation_date', 'last_update_date'];
```

Las relaciones que hemos definido en el modelo PointOfInterest son las siguientes:

- **Usuario →** un punto de interes sera creado y actualizado por un usuario.
- **Punto de interés →** una fotografía ha de tener un punto de interés.
- **Áreas temáticas →** uno o varios puntos de interés puede tener varias áreas temáticas.
- **Fotografías →** un punto de interés puede tener muchas fotografías.
- **Lugar →** un punto de interés tendrá asignado un lugar.
- **vídeos →** un punto de interés puede tener muchos vídeos también.
- **Visitas →** un punto de interés podría tener varias visitas.

```php
public function userCreator()
{
    return $this->belongsTo(User::class, 'creator');
}

public function userUpdater()
{
    return $this->belongsTo(User::class, 'updator');
}

public function thematicAreas()
{
    return $this->belongsToMany(ThematicArea::class)->withPivot('point_of_interest_id', 'title', 'description', 'language');
}

public function photographies()
{
    return $this->hasMany(Photography::class);
}

public function place()
{
    return $this->belongsTo(Place::class);
}

public function videos()
{
    return $this->hasMany(Video::class);
}

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

Siguiendo el modelo, hemos desarrollado lo siguiente:

- El método estático **create()**, donde cada vez que registremos un nuevo punto de interés, guardaremos el id de usuario creado con su fecha de creación. Además, insertamos una url para un punto de interés registrado.
- Disparadores a la hora de:
    - **Actualizar →** actualiza el id de usuario editado y la fecha de edición.
    - **Borrar →** el punto de interés guardado en las tablas fotografías, vídeos y visitas, se cambiaran a valor **null**.

```php
public static function create(array $attributes = [])
{
    $attributes['creation_date']=Carbon::now();
    $attributes['creator']= auth()->user()->id;

    $pointOfInterest = static::query()->create($attributes);
    $pointOfInterest->generateSlug();

    return $pointOfInterest;
}

public static function boot()
{
    parent::boot();

    static::updating(function($pointsofinterest) {
        $pointsofinterest->last_update_date = Carbon::now();
        $pointsofinterest->updater = auth()->user()->id;
    });

    static::deleting(function($pointOfInterest){
        $pointOfInterest->thematicAreas()->detach();
        $pointOfInterest->photographies()->each(function($p) {
            $p->point_of_interest_id = null;
            $p->save();
        });

        $pointOfInterest->visits()->each(function($v) {
            $v->point_of_interest_id = null;
            $v->save();
        });

        $pointOfInterest->videos()->each(function($v) {
            $v->point_of_interest_id = null;
            $v->save();
        });
    });
}
```

Definimos un método para el registro y actualización de los campos de la tabla pivote **puntos_de_interes_area_tematica**. Borramos las filas y, si no existe el id del área temática coincidente con el id de punto de interés (desarrollado en otro método), entonces registramos una fila con los datos de los campos: **título, descripción** e **idioma**. En caso contrario, se actualizarán las filas con los campos adicionales.

```
public function syncthematicAreas($thematicAreas, $title, $description, $language)
{
    $this->thematicAreas()->detach();

    if(!$this->existThematicAreaId($thematicAreas)) {
        $this->thematicAreas()->attach($thematicAreas, [
            'title' => $title,
            'description' => $description,
            'language' => $language
        ]);
    }

    return $this->thematicAreas()->updateExistingPivot($thematicAreas, [
        'title' => $title,
        'description' => $description,
        'language' => $language
    ]);
}

private function existThematicAreaId($id)
{
    return $this->thematicAreas()
        ->where('thematic_area_id', '=', $id)
        ->exists();
}
```

A continuación, tenemos desarrollado lo siguiente:

- Un método donde registramos el slug a la hora de guardar un punto de interés, comprobando que no sea el mismo.
- Recibiremos como parámetro de acceso el propio slug o url.
- Una consulta indicando si el usuario puede ver todos los puntos de interés. Si no tiene permiso, puede ver sus propios puntos de interés siempre que sea profesor.

```php
public function generateSlug()
{
  $url = Str::slug($this->qr);

  if(static::whereUrl($url)->exists()) {
    $url .= '--' . static::where('url', 'like', $url . '-%')->count();
  }

  $this->url = $url;
  $this->save();
}

public function getRouteKeyName()
{
  return 'url';
}

public function scopeAllowed($query)
{
  if(auth()->user()->can('view', $this)) {
    return $query;
  }else{
    if(auth()->user()->hasRole('Profesor')){
      return $query->where('creator', auth()->id())->orWhere('updater', auth()->id());
    }
    abort(403);
  }
}
```

Para terminar, dos métodos relacionados con las estadísticas:

- El primero devuelve el número de los puntos de interés creados durante las últimas 24 horas.
- El segundo contabiliza los puntos de interés creados durante los últimos 7 días.

```php
public static function countNewPointsOfInterest()
{
    return (int)count(PointOfInterest::whereDate('creation_date', Carbon::today())->get());
}

public static function datesForGrafic(){
      return PointOfInterest::query()->where('deleted_at','=',null)
      ->whereDate('creation_date','>=',
      Carbon::now()->subDays(7))->get()->groupBy(function($date) {
            return Carbon::parse($date->creation_date)->format('d-m-Y');
});
}
```

Clase Visit

Seguimos con la clase **Visit**, donde hemos añadido estas mismas clases ya usadas en las anteriores.

```php
<?php

namespace App;

use Carbon\Carbon;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;
use Illuminate\Support\Str;
```

Los campos que se le han realizado una asignación masiva de datos son los siguientes: **hour**, **deviceid**, **url**, **appversion**, **useragent**, **ssoo**, **ssooversion**, **latitude**, **longitude** y **point_of_interest_id**. El campo de tipo **database** es **hour**.
Solo contamos con una relación:

- **Punto de interés →** una visita tiene que tener un punto de interés.

```
class Visit extends Model
{
  use SoftDeletes;

  protected $fillable = ['hour', 'deviceid', 'url', 'appversion', 'useragent', 'ssoo', 'ssooversion', 'latitude', 'longitude', 'point_of_interest_id'];
  protected $dates = ['hour'];

  public function point_of_interest()
  {
    return $this->belongsTo(PointOfInterest::class);
  }
```

Definimos el método estático **create()** ya que lo necesitamos para insertar una url de una visita registrada.

```
public static function create(array $attributes = [])
{
  $visit = static::query()->create($attributes);

  $visit->generateSlug();

  return $visit;
}
```

Siguiendo este modelo, tenemos desarrollado lo siguiente:

- Un método donde registramos el slug o url al registrar en una nueva fila una visita, comprobando si el slug existía o no.
- Un mutator donde registramos el formato de la hora de visita.
- Recibiremos como parámetro de acceso el propio slug.

```
  public function generateSlug()
  {
    $url = Str::slug($this->deviceid);

    if(static::whereUrl($url)->exists()) {
      $url .= '--' . static::where('url', 'like', $url . '-%')->count();
    }

    $this->url = $url;
    $this->save();
  }

  public function setHourAttribute($hour)
  {
     $this->attributes['hour'] = $hour ? Carbon::parse($hour)->format('Y-m-d H:i:s') : null;
  }

  public function getRouteKeyName()
  {
     return 'url';
  }
}
```

Completando el modelo, hemos desarrollado una serie de métodos relacionados con las estadísticas:

- El primer método devuelve un número de visitas creadas durante los últimos 7 días.
- El segundo método devuelve una lista con los 5 primeros puntos de interés más visitados.

```
public static function DatesForGrafic()
{
  return Visit::query()->whereDate('hour','>=', Carbon::now()->subDays(7))->get()
    ->groupBy(
      function($date) {
        return Carbon::parse($date->hour)->format('d-m-Y' );
      });
}

public static function getPointsOfInterestMostVisit(){
  return Visit::query()->get()->groupBy('point_of_interest_id')->take(5)->sort();
}
```

Clase ThematicArea

El siguiente modelo que hemos desarrollado es la clase **ThematicArea**. Unicamente, solo hemos utilizado estas clases:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;
use Illuminate\Support\Str;
```

Los campos a los que se le realizaran una asignación masiva son: **name** y **description**. Las relaciones para este modelo son:

- **Puntos de interés →** una o varias áreas temáticas puede tener uno o varios puntos de interés.
- **Usuarios →** un área temática puede asignarse a varios usuarios.
- **Fotografías →** un área temática puede asignarse también a varias fotografías.
- **vídeos →** un área temática puede asignarse a muchos vídeos.

```php
class ThematicArea extends Model
{
  use SoftDeletes;

  protected $fillable = ['name','description'];

  public function pointsOfInterest(){
    return $this->belongsToMany(PointOfInterest::class)->withPivot('thematic_area_id', 'title', 'description', 'language');
  }

  public function photographies()
  {
    return $this->hasMany(Photography::class);
  }

  public function videos()
  {
    return $this->hasMany(Video::class);
  }

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

Definimos el método **create()** donde llamaremos a un método que se encarga de generar un slug o url para un área temática.
Hemos desarrollado los disparadores. Solo a la hora de borrar, ya que el área temática asignada en los usuarios, si tiene, fotografías y vídeos se les cambia los valores a **null**.

```php
  public static function create(array $attributes = [])
  {
    $thematic_area = static::query()->create($attributes);
    $thematic_area->generateSlug();

    return $thematic_area;
  }

  public static function boot()
  {
    parent::boot();

    static::deleting(function($thematicarea) {
      $thematicarea->pointsOfInterest()->detach();
      $thematicarea->users()->each(function ($u){
        $u->thematic_area_id = null;
        $u->save();
      });

      $thematicarea->photographies()->each(function ($p){
        $p->thematic_area_id = null;
        $p->save();
      });

      $thematicarea->videos()->each(function ($v){
        $v->thematic_area_id = null;
        $v->save();
      });
    });
  }
```

Completando el modelo, hemos desarrollado:

- Un método registrando un slug, comprobando si existe previamente o no.
- Recibiremos como parámetro de acceso el propio slug o url.

```
  public function generateSlug()
  {
    $url = Str::slug($this->name);

    if(static::whereUrl($url)->exists()) {
       $url .= '--' . static::where('url', 'like', $url . '-%')->count();
    }

    $this->url = $url;
    $this->save();
  }

  public function scopeAllowed($query)
  {
     if(auth()->user()->can('view', $this)) {
        return $query;
     }
  }

  public function getRouteKeyName()
  {
     return 'url';
  }
}
```

Clase Video

La clase **Video** es el siguiente modelo que hemos desarrollado. Estas son las clases que hemos usado:

```
<?php

namespace App;

use Carbon\Carbon;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;
use Illuminate\Support\Facades\Request;
use Illuminate\Support\Facades\Storage;
use Illuminate\Support\Str;
```

En esta clase, contamos con los campos que se les realizan una asignación masiva: **name**, **url**, **route**, **point_of_interest_id**, **order**, **published**, **date_create**, **last_update**, **creator**, **updater**, **thematic_area_id**. Además, definimos los campos de tipo **database**: **date_create** y **last_update**.

```
class Video extends Model
{
  use SoftDeletes;

  protected $fillable = ['name', 'url', 'route', 'point_of_interest_id', 'order', 'published', 'date_create', 'last_update',
'creator', 'updater', 'thematic_area_id'];
  protected $dates = ['date_create', 'last_update'];
```

Las relaciones en este modelo son las siguientes:

- **Elementos vídeo** → un mismo vídeo puede tener varias características de vídeo.
- **Usuario** → un vídeo sera creado y actualizado por un usuario.
- **Punto de interés** → un vídeo puede tener asignado un punto de interés.
- **Área temática** → un vídeo pertenece a un área temática.

```php
public function video_items()
{
    return $this->hasMany(VideoItem::class);
}

public function userCreator()
{
    return $this->belongsTo(User::class, 'creator');
}

public function userUpdater()
{
    return $this->belongsTo(User::class, 'updater');
}

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

public function thematic_area()
{
    return $this->belongsTo(ThematicArea::class);
}
```

Avanzando con el modelo, hemos desarrollado lo siguiente:

- El método estático **create()**, donde cada vez que registremos un nuevo vídeo, guardamos el id de usuario creado con su fecha de creación. Además, registramos un slug o una url.
- Disparadores a la hora de:
    - **Actualizar →** actualiza el id de usuario editado, la fecha de edición y, si hay vídeo, lo vamos a guardar en formato .mp4 en una carpeta Storage llamada **videos.**.
    - **Borrar →** borramos el vídeo almacenado de la carpeta **vídeos** en Storage.

```php
public static function boot()
{
    parent::boot();

    static::updating(function($video) {
        if(Request::has('route')) {
            $file_name = $video->url . '.mp4';
            $video->route = Request::file('route')->storeAs('public/videos', $file_name);
        }
        $video->updater = auth()->user()->id;
        $video->last_update = Carbon::now();
    });

    static::deleting(function($video){
        Storage::disk('public')->delete($video->route);
        $video->video_items()->delete();
    });
}

public static function create(array $attributes = [])
{
    $attributes['creator'] = auth()->user()->id;
    $attributes['date_create'] = Carbon::now();

    $video = static::query()->create($attributes);
    $video->generateSlug();

    return $video;
}
```

Además, hemos desarrollado lo siguiente:

- Un método donde registra el slug, comprobando si existe previamente.

- Recibiremos como parámetro de acceso el propio slug.
- Una doble consulta indicando que si puede ver todos los vídeos. Si cumple la condición, en caso de que tenga el rol profesor, puede ver solo los vídeos relacionados con su departamento. En caso contrario, solo vería sus propios vídeos.

```
public function generateSlug()
{
  $url = Str::slug($this->name);

  if(static::whereUrl($url)->exists()) {
    $url .= '--' . static::where('url', 'like', $url . '-%')->count();
  }

  $this->url = $url;
  $this->save();
}

public function getRouteKeyName()
{
  return 'url';
}

public function scopeAllowed($query)
{
  if(auth()->user()->can('view', $this)) {
    if (auth()->user()->hasRole('Profesor')){
          return $query->where('thematic_area_id', auth()->user()->thematic_area_id);
        }
    return $query;
  }else{
    return $query->where('creator', auth()->id())->orWhere('updater', auth()->id());
  }
}
```

Para terminar, declaramos un método que devuelve el número de vídeos registrados durante las últimas 24 horas.

```
  public static function countNewVideos()
  {
    return (int)count(Video::whereDate('created_at', Carbon::today())->get());
  }
}
```

Clase VideoItem

Terminamos con la clase **VideoItem** donde añadimos las clase necesarias. Después, los campos de asignación masiva que son: **video_id**, **url**, **quality**, **format**, **orientation** y **language**. Solo contamos con una relación, que es la siguiente:

- **vídeos →** unas características de vídeo puede tenerlas en varios vídeos.

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\SoftDeletes;
use Illuminate\Support\Str;

class VideoItem extends Model
{
  use SoftDeletes;

  protected $fillable = ['video_id', 'url', 'quality', 'format', 'orientation', 'language'];

  public function video()
  {
    return $this->belongsTo(Video::class);
  }
```

Avanzando con el modelo, tenemos desarrollo:

- Un disparador ya que a la hora de crear una nueva fila, obtenemos el id del vídeo.
- El método estático **create()** se encarga de generar el slug de este modelo a través de un método.

```
public static function boot()
{
    parent::boot();

    static::creating(function($videoItem){
        $videoItem->video_id = $videoItem->video->id;
    });
}

public static function create(array $attributes = [])
{
    $videoItem = static::query()->create($attributes);

    $videoItem->generateSlug();

    return $videoItem;
}
```

En el siguiente método, generamos el slug comprobando si existe. Por último, accederemos al modelo de acceso el slug o url como parámetro.

```
public function generateSlug()
{
    $url = Str::slug($this->video->name);

    if(static::where('url', $url)->exists()) {
        $url .= '--' . static::where('url', 'like', $url . '-%')->count();
    }

    $this->url = $url;
    $this->save();
}

public function getRouteKeyName()
{
    return 'url';
}
}
```

Controladores

En este apartado, hemos documentado algunos de los controladores que contiene un código más interesante o importante a la hora de explicar. Los controladores son los siguientes:

LoginController

En **LoginController** hemos implementado una serie de métodos a la hora de validar el usuario, ya que vamos a comprobarlo por correo o nombre de usuario.
Lo primero va a ser definir un atributo que va a ser el usuario que se va a loguear. Ese atributo lo definimos en un constructor donde va a llamar a un método.

```php
<?php

namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use App\Providers\RouteServiceProvider;
use Illuminate\Foundation\Auth\AuthenticatesUsers;

class LoginController extends Controller
{

  use AuthenticatesUsers;

  protected $redirectTo = RouteServiceProvider::HOME;

  protected $user;

  public function __construct()
  {
     $this->middleware('guest')->except('logout');
     $this->user = $this->findLoginName();
  }
```

El siguiente método recibe un usuario del formulario de login ya que no sólo valida el usuario sino que, además, comprueba desde la base de datos si el usuario se encuentra activo. Si es cierto, pasa esa validación, pero tiene que comprobar si existen esas credenciales para acceder al panel de control. En caso contrario, regresa al login mostrando un mensaje de error.

```
public function login(\Illuminate\Http\Request $request) {
    $this->validateLogin($request);

    if ($this->hasTooManyLoginAttempts($request)) {
        $this->fireLockoutEvent($request);
        return $this->sendLockoutResponse($request);
    }

    if ($this->guard()->validate($this->credentials($request))) {
        $user = $this->guard()->getLastAttempted();

        if ($user->active && $this->attemptLogin($request)) {
            return $this->sendLoginResponse($request);
        } else {
            $this->incrementLoginAttempts($request);
            return redirect()
                ->back()
                ->withInput($request->only($this->username(), 'remember'))
                ->withErrors(['active' => 'Cuenta desactivada,por favor ponte en contacto de algún administrador.']);
        }
    }

    $this->incrementLoginAttempts($request);

    return $this->sendFailedLoginResponse($request);
}
```

Completando el desarrollo del controlador, hemos desarrollado un primer método donde obtenemos un valor del campo de un formulario de login ya que comprueba si está iniciando sesion con un correo. Si no inicia sesion con un correo, entonces se iniciara con un nombre de usuario.
El segundo método devuelve el valor del atributo del usuario que está logueado.

```
public function findLoginName()
{
    $user = request()->input('user');
    $field = filter_var($user, FILTER_VALIDATE_EMAIL) ? 'email' : 'login';
    request()->merge([$field => $user]);
    return $field;
}

public function username()
{
    return $this->user;
}
}
```

AdminController

El controlador **AdminController** envía a la vista una serie de datos relacionado con estadisticas y graficas con el fin de mostrar al usuario la información a medida que vaya administrando el dashboard.
Los datos que vamos a enviar a la vista son los siguientes:

- Cuantos usuarios se han registrado en un dia.
- Cuantas fotos y vídeos se han creado en un dia.
- Cuantos puntos de interés se han creado en un dia.
- Cuántas visitas se han dado de alta durante los últimos 7 días diseñado en una gráfica.
- Cuantos puntos de interés se han creado durante los últimos 7 días en una gráfica.

- Un ranking con los 5 puntos de interés más visitados.

```php
<?php

namespace App\Http\Controllers\Admin;

use App\Charts\GraphicChart;
use App\Photography;
use App\PointOfInterest;
use App\User;
use App\Video;
use App\Visit;
use Carbon\Carbon;
use App\Http\Controllers\Controller;

class AdminController extends Controller
{
  public function index()
  {
    $numberUsers = User::countNewUsers();
    $photos = Photography::countNewPhotos();
    $videos = Video::countNewVideos();
    $numberPointsOfInterest =PointOfInterest::countNewPointsOfInterest();

    $visits = Visit::datesForGrafic();
    $chartVisit = $this->createChart($visits, 'Visitas', 'line', 'rgba(255, 99, 132, 0.2)', '#F96332');

    $pointsOfInterest = PointOfInterest::datesForGrafic();
    $chartPointOfInterest = $this->createChart($pointsOfInterest, 'Punto de intereses', 'bar', 'rgba( 206, 73, 0, 0.4)', '#F96332');

    $mostVisits = Visit::getPointsOfInterestMostVisit();
    $pointsOfInterestMostVisits = $this->getPointsOfInterest($mostVisits);

    return view('admin.dashboard', compact(['numberUsers', 'pointsOfInterestMostVisits', 'numberPointsOfInterest', 'chartVisit',
'chartPointOfInterest', 'photos', 'videos']));
  }
```

Para la creación de las gráficas, hemos desarrollado un método que recibe un nombre de la gráfica, el tipo de gráfica y los estilos a diseñar. Devolvemos los valores con el fin de crear una gráfica con los datos y con los estilos definidos.

```php
  private function createChart($data, $name, $type, $backgroundColor, $color)
  {
    $valores = $this->getValuesFromArray($data);
    $chart = new GraphicChart();
    $chart->displayLegend(false);
    $chart->labels(array_reverse($valores[0]));
    $chart->dataset($name, $type, array_reverse($valores[1]))->backgroundcolor($backgroundColor)->color($color);
    return $chart;
  }
```

Además, hemos desarrollado otro método donde devuelve en un array los valores de punto de interés contabilizados, separando las fechas por un lado y un contador por el otro.

```php
  private function getValuesFromArray($array)
  {
    $i = 0;
    $result = [];
    foreach ($array as $key => $value) {
      $result[0][$i] = $key;
      $result[1][$i] = count($value);
      $i++;
    }
    return $result;
  }
}
```

En la parte del ranking con los puntos de interés más visitados, hemos definido un método donde devuelve una lista de los puntos de interés encontrados, consultando con la base de datos.

```php
private function getPointsOfInterest($array)
{
    $i = 0;
    $result = [];
    foreach ($array as $key => $value) {
        $result[] = PointOfInterest::find($key);
    }

    return array_reverse($result);
}
```

UsersController

El controlador **UsersController**, donde no solo hemos desarrollado los métodos fundamentales para la administración de usuarios. Además, cuenta con la funcionalidad basada en la actualización de fotos de perfil y estadísticas de usuarios.
Recordando aquellos métodos que componían un controlador de tipo recurso, el método **index()** listaba todos los usuarios, comprobando si requiere permiso para ver los usuarios.

```php
<?php

namespace App\Http\Controllers\Admin;

use App\Events\AdminInformed;
use App\Events\UserWasRegisted;
use App\Http\Controllers\Controller;
use App\Http\Requests\UpdateUsersRequest;
use App\Photography;
use App\Place;
use App\ThematicArea;
use App\User;
use App\Video;
use Illuminate\Http\Request;
use Illuminate\Support\Str;
use Spatie\Permission\Models\Permission;
use Spatie\Permission\Models\Role;
use Hashids\Hashids;

class UsersController extends Controller
{
    public function index()
    {
        $users = User::allowed()->get();
        return view('admin.users.index', compact('users'));
    }
```

El método **create()** enviaba a una vista donde teníamos un formulario de creación con los datos necesarios para los campos de selección de un formulario. Además, puede acceder a la vista si tiene permiso de crear usuarios.

```php
public function create()
{
    $user = new User;
    $this->authorize('create', $user);
    $thematic_areas = ThematicArea::all();
    $roles = Role::with('permissions')->get();
    $permissions = Permission::pluck('name', 'id');
    return view('admin.users.create', compact('thematic_areas', 'roles', 'permissions', 'user'));
}
```

El método **store()** enviaba los datos del formulario y los validaba siempre que tenga un permiso. Lo que nos vamos a centrar más en este método es que una vez que todo hay ido bien en el registro, hemos desarrollado lo siguiente:

- Vamos a generar una contraseña aleatoria de 8 caracteres con el método **Str::random(8)**.
- Asignamos los roles y permisos (si se ha seleccionado) a ese usuario.
- Un primer método **UserWasRegistered::dispatch()** donde recibe el usuario registrado y la contraseña aleatoria útiles para el reenvío de un correo electrónico a ese usuario registrado como evento.
- Un segundo método **AdminInformed::dispatch()** donde recibe los correos de los administradores, obtenidos en un método, y el usuario registrado ya que vamos a enviar a cada administrador un correo de notificación también como evento.

```php
public function store(Request $request)
{
    $this->authorize('create', new User);
    $data = $this->validate($request, [
        'login' => 'required | unique:users',
        'email' => 'required | string | unique:users',
        'name' => 'required',
        'surnames' => 'required'
    ], [
        'login.required' => 'El nombre de usuario es requerido',
        'login.unique' => 'El nombre de usuario ya existe en nuestros registros',
        'email.required' => 'El correo electronico es requerido',
        'email.string' => 'El correo no es valido',
        'email.email' => 'El correo no es valido',
        'email.unique' => 'El correo electronico ya existe en nuestros registros',
        'name.required' => 'El nombre es requerido',
        'surnames.required' => 'El campo apellidos es requerido',
    ]);

    $data['password'] = Str::random(8);
    $user = User::create($data);

    $user->assignRole($request->roles);
    $user->givePermissionTo($request->permissions);

    $admins = $this->getAdmins();

    UserWasRegisted::dispatch($user, $data['password']);
    AdminInformed::dispatch($admins, $user);

    return redirect()->route('admin.users.index')->with('flash', 'El usuario ha sido creado correctamente');
}
```

Para obtener todos los administradores del panel de control, implementamos un método que devuelve en un array los administradores localizados y encontrados en la base de datos.

```
private function getAdmins()
{
   $role = Role::findByName('Administrador');
   $admins = $role->users()->pluck('email');
   return $admins;
}
```

El método **show()** enviaba a la vista el contenido de un usuario. Un detalle importante es que vamos a ocultar el id a través de un proceso de encriptación. Para ello, utilizamos la librería **hashids** donde codificamos el id con el fin de ocultar su valor.

```
public function show(User $user)
{
   $this->authorize('view', $user);
   $hashids = new Hashids('secret',7);
   $idEncript=$hashids->encode($user->id);

   return view('admin.users.show', compact('user','idEncript'));
}
```

El método **edit()** enviaba los datos del modelo a una vista de un formulario con los roles y permisos asignados. Cómo recibe un id como parámetro, lo codificamos aplicando la librería **encode()**. Para llevar a cabo la implementación, tiene que tener un permiso.

```
public function edit(User $user)
{
   $this->authorize('update', $user);
   $thematic_areas = ThematicArea::all();
   $roles = Role::with('permissions')->get();
   $permissions = Permission::pluck('name', 'id');
   $hashids = new Hashids('secret',7);
   $idEncript=$hashids->encode($user->id);

   return view('admin.users.edit', compact('user', 'thematic_areas', 'permissions', 'roles','idEncript'));
}
```

El método **update()** actualizaba los datos enviados y validados desde el formulario si tenía permiso para editar. Comprobamos si el usuario se encuentra marcado como activo o inactivo. Devuelve una redirección alternativa:

- Si un usuario es alumno o profesor, redirige a la misma página.
- Si un usuario es admin o superadmin, redirige al listado de los usuarios.

```php
public function update(User $user, UpdateUsersRequest $request)
{
    $this->authorize('update', $user);
    ( $request->get('active')=='1' ) ? $request->active = 1 : $request->active = 0;

    $user->update($request->validated());

     if(auth()->user()->roles->first()->name == "Alumno" || auth()->user()->roles->first()->name == "Profesor") {
        return back()->with('flash', 'Tus datos han sido actualizados correctamente');
     }

    return redirect()
        ->route('admin.users.index', $user)
        ->with('flash', 'El usuario ' . $user->login . ' ha sido actualizado correctamente');
}
```

Y el método **destroy()**, borraba un usuario. Pero hay un aspecto a tener en cuenta. Si un usuario es superadmin, no puede ser eliminado porque tiene el rol **Super Administrador**. Ahora, como admin, si borra a un usuario que también es admin, mostramos un error diciendo que el usuario es admin y, así, evitamos el borrado de los administradores.

```php
public function destroy(User $user)
{
    $this->authorize('delete', $user);
    if ( $user->hasRole('Super Administrador') || $user->hasRole('Administrador') ) {
        return back()->with('danger', 'El usuario ' . $user->login . ' no ha podido ser eliminado, ya que no tiene permisos para ello');
    }

    if(auth()->user()->roles->first()->name == "Administrador") {
        if ($user->hasRole('Administrador')) {
            return back()->with('danger', 'El usuario ' . $user->login . ' no ha podido ser eliminado, ya que tiene el rol Administrador');
        }
    }

    $user->delete();

    return redirect()->route('admin.users.index')->with('flash', 'El usuario ' . $user->login . ' ha sido eliminado correctamente');
}
```

Ahora pasamos a la funcionalidad de la foto de perfil a la hora de actualizar. En el método **updatePhoto()** recibe una imagen subida por Ajax implementada por la librería **Croppie**. En un método, procesamos la foto. Luego, a esa foto la almacenamos en una carpeta de Storage llamada **users** y, a continuación, devolvemos un mensaje en formato JSON como que todo ha ido bien.

```php
public function updatePhoto(Request $request)
{
    $hashids = new Hashids('secret',7);
    $id= $hashids->decode($request->get('user'));
    $user = User::find((int)$id[0]);
    $this->authorize('update', $user);
    $data = $request->get('profile');

    $data = $this->processPhoto($data);

    file_put_contents(public_path().'/storage/users/' . $data[1] , $data[0]);

    $user->update([
        'profile' => "users/" . $data[1]
    ]);

    return response()->json(['success' => 'success']);
}
```

Para procesar la foto, declaramos un método donde recibimos la foto subida. Con esa foto, primero, la codificamos en base64 y, después, nombramos la imagen en forma de fecha de hoy en segundos en formato .png. Devuelve esa foto procesada.

```php
public function processPhoto($data)
{
    list($type, $data) = explode(';', $data);
    list(, $data) = explode(',', $data);
    $result[0] = base64_decode($data);
    $result[1] = time() . '.png';
    return $result;
}
```

Este método está relacionado con las estadísticas. El método **countStatisticsUsers()** enviamos a la vista del dashboard principal cuantas fotos y vídeos ha registrado un usuario.

```php
public function countStatisticsUsers()
{
    $photographies = Photography::get()->Auth::user();
    $videos = Video::get()->Auth::user();

    return view('admin.dashboard', compact(['photographies', 'videos']));
}
```

Por último, hemos definido un método que consiste en mostrar u ocultar el checkbox **Super Administrador** en la vista de los checkboxes de roles. Para ello, autorizaremos a aquel usuario que tenga ese rol. Ese usuario tendrá habilitado el checkbox tanto en la vista **users/create.blade.php** como **users/edit.blade.php**.

```php
public function manage(Request $request)
{
    $request->user()->authorizeRoles('Super Administrador');

    $authSuperAdmin = Auth::user()->hasRole('Super Administrador');
    $users = User::all();
    $users->reject($authSuperAdmin);

    return view('admin.roles.checkboxes')->with('users', $users);
  }
}
```

Validaciones

Como en la mayoría de los controladores teníamos desarrollados más de dos reglas de validación en los métodos de **update()**, hemos creado unas clases de tipo Request no solo para reducir código en los controladores sino para ubicar donde tenemos definidas realmente las reglas de validación con sus mensajes. Estas clases se crean ejecutando el comando **php artisan make:request [nombre_claseRequest]**.

UpdateUsersRequest

Comenzamos con la clase **UpdateUsersRequest** donde definimos una serie de reglas a la hora de validar un usuario cuando lo estamos actualizando. Validaremos los siguientes campos:

- El nombre de usuario (no puede ser repetido) debe ser obligatorio.
- Su nombre y sus apellidos deben ser requeridos.
- Si introduce contraseña, que tenga como mínimo 6 caracteres y confirme la contraseña correctamente.
- Es necesario validar también la área temática y la activación del usuario.

```php
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;
use Illuminate\Validation\Rule;

class UpdateUsersRequest extends FormRequest
{
  public function authorize()
  {
    return true;
  }

  public function rules()
  {
    $rules = [
      'login'=> [
        'required',
        Rule::unique('users')->ignore($this->route('user')->id)
      ],
      'name' => 'required',
      'surnames' => 'required',
      'thematic_area_id' => 'not_in:0'
    ];

    if ($this->filled('password')) {
      $rules['password'] = 'confirmed | min:6';
    }

    if ($this->filled('active')) {
     $rules['active'] = 'required';
   }

    return $rules;
  }
```

Cuando estemos actualizando un usuario y no cumple alguna validación, esa validación va a tener su mensaje de error personalizado en el método **messages()**:

```
  public function messages()
  {
     return [
        'login.required' => 'El nombre de usuario es requerido',
        'login.unique' => 'El nombre de usuario ya existe en nuestros registros',
        'password.confirmed' => 'Ambas contraseñas son distintas',
        'password.min' => 'Contraseña corta, debe tener como minimo 6 caracteres',
        'name.required' => 'El campo nombre es requerido',
        'surnames.required' => 'El campo apellidos es requerido',
     ];
  }
}
```

UpdatePhotographiesRequest

Con la clase **UpdatePhotographiesRequest** definimos un conjunto de reglas a la hora de validar una foto cuando la estamos actualizando. Los siguientes campos que vamos a validar son:

- El nombre de la foto es requerido y con un tamaño máximo de 100 caracteres.
- El orden de la foto será obligatorio.
- Debe seleccionar un punto de interés para la foto.
- Debe seleccionar un área temática para la foto.
- Si sube una foto, la foto será de tipo imagen cuyo tamaño máximo es de 3 MB.

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class UpdatePhotographiesRequest extends FormRequest
{
  public function authorize()
  {
     return true;
  }

  public function rules()
  {
     $rules = [
        'name' => 'required | max: 100',
        'order' => 'required',
        'point_of_interest_id' => 'required',
        'thematic_area_id' => 'required',
     ];

     if($this->has('route')) {
        $rules['route'] = 'image | max: 3072';
     }

     return $rules;
  }
```

Cuando actualicemos una foto y nos muestre un error de validación, se mostraran estos mensajes que devolverán desde el método **messages()**.

```
  public function messages()
  {
    return  [
      'name.required' => 'El nombre de la foto es requerido',
      'name.max => 'Tamaño máximo para el nombre de la foto es de 100 caracteres',
      'route.image' => 'La foto debe ser una imagen',
      'route.max' => 'El tamaño máximo de la imagen es de 3 MB',
      'order.required' => 'El número de orden de la foto es requerido',
      'point_of_interest_id.required' => 'Debe seleccionar un punto de interés',
      'thematic_area_id.required' => 'Debe seleccionar un área temática'
    ];
  }
}
```

UpdatePointOfInterestRequest

Para los puntos de interés, contamos con la clase **UpdatePointOfInterestRequest**, cuyas reglas que vamos a validar a la hora de actualizar un punto de interés son las siguientes:

- El código qr es requerido con tamaño máximo de 150 caracteres.
- La distancia, la latitud y la longitud para el punto de interés es obligatorio.
- Debemos seleccionar un lugar para el punto de interés
- También es obligatorio seleccionar alguna área temática.
- El título que vamos a insertar va a ser requerido.
- La descripción es requerida con 200 caracteres como tamaño máximo y el idioma es requerido con 5 caracteres como máximo.

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class UpdatePointsOfInterestRequest extends FormRequest
{
  public function authorize()
  {
    return true;
  }

  public function rules()
  {
    return [
      'qr' => 'required | max: 100',
      'distance' => 'required',
      'latitude' => 'required',
      'longitude' => 'required',
      'place_id' => 'required',
      'thematicAreas' => 'required',
      'title' => 'required',
      'description' => 'required | max: 200',
      'language' => 'required | max: 5',
    ];
  }
```

Si no cumplimos con algunas de las validaciones a la hora de actualizar un punto de interés, contaremos con estos mensajes de error desarrollados en el método **messages()**.

```
public function messages()
{
    return [
        'qr.required' => 'El codigo qr es requerido',
        'qr.required' => 'Tamaño codigo qr invalido. Valor maximo 150',
        'distance.required' => 'La distancia es requerida',
        'latitude.required' => 'La latitud es requerida',
        'longitude.required' => 'La longitud es requerida',
        'place_id.required' => 'Debe seleccionar un lugar',
        'thematicAreas.required' => 'La seleccion de area tematica es requerida',
        'title.required' => 'El campo titulo debe ser requerido',
        'description.required' => 'El campo descripcion debe ser requerido',
        'description.max' => 'Tamaño maximo 150 caracteres',
        'language.required' => 'El campo idioma debe ser requerido',
        'language.max' => 'Tamaño maximo 150 caracteres',
    ];
  }
}
```

UpdateVisitsRequest

La siguiente clase es **UpdateVisitsRequest**, donde definimos las validaciones cuando actualicemos una visita. Los campos que vamos a validar son los siguientes:

- El nombre de dispositivo es obligatorio con un tamaño máximo de 85 caracteres.
- La versión del dispositivo es también obligatorio con 45 caracteres como tamaño máximo.
- Lo mismo con el usuario del dispositivo con tamaño máximo de 95 caracteres.
- La hora del dispositivo es obligatoria.
- El SSOO y su versión son obligatorios con un tamaño máximo de 45 caracteres para ambos.
- La longitud y la latitud son requeridos.
- Debe tener un punto de interés seleccionado.

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class UpdateVisitsRequest extends FormRequest
{
  public function authorize()
  {
    return true;
  }

  public function rules()
  {
    return [
      'deviceid' => 'required | max:85',
      'appversion' => 'required | max:45',
      'useragent' => 'required | max:95',
      'hour' => 'required',
      'ssoo' => 'required | max:45',
      'ssooversion' => 'required | max:45',
      'latitude' => 'required',
      'longitude' => 'required',
      'point_of_interest_id' => 'required'
    ];
  }
```

A la hora de actualizar una visita, nos muestra un error al no cumplir con una de las reglas de validación, se mostrará uno de estos mensajes definidos en el método **messages()**.

```
public function messages()
{
    return [
        'deviceid.required' => 'El campo nombre de dispositivo es requerido',
        'deviceid.max' => 'Tamaño maximo 85 caracteres',
        'appversion.required' => 'El campo version de la app es requerido',
        'appversion.max' => 'Tamaño maximo 45 caracteres',
        'useragent.required' => 'El campo usuario de dispositivo es requerido',
        'useragent.max' => 'Tamaño maximo 95 caracteres',
        'hour.required' => 'El campo hora es requerido',
        'ssoo.required' => 'El campo Sistema Operativo es requerido',
        'ssoo.max' => 'Tamaño maximo 45 caracteres',
        'ssooversion.required' => 'El campo version de sistema operativo es requerido',
        'ssooversion.max' => 'Tamaño maximo 45 caracteres',
        'latitude.required' => 'El campo latitud es requerido',
        'longitude.required' => 'El campo longitud es requerido',
        'point_of_interest_id.required' => 'Debe seleccionar un punto de interés',
    ];
  }
}
```

UpdateVideosRequest

Para los vídeos, hemos creado una clase llamada **UpdateVideosRequest** donde hemos definido las siguientes reglas de validación a la hora de actualizar un vídeo:

- El nombre del vídeo es obligatorio y con un tamaño máximo de 100 caracteres.
- La enumeración del vídeo va a ser obligatoria.
- Debe seleccionar un punto de interés para el vídeo.
- Debe seleccionar un área temática para el vídeo.
- Si sube un vídeo, el vídeo tiene que tener un formato .mp4, .webm o mpeg, con un tamaño maximo de 100 MB.

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class UpdateVideosRequest extends FormRequest
{
  public function authorize()
  {
    return true;
  }

  public function rules()
  {
    $rules = [
      'name' => 'required | max:100',
      'order' => 'required',
      'point_of_interest_id' => 'required',
      'thematic_area_id' => 'required'
    ];

    if($this->has('route')) {
      $rules['route'] = 'mimes:mpeg,mp4,webm | max:100000';
    }
    return $rules;
  }
```

Al no cumplir una validación cuando estamos actualizando un vídeo, se mostrarán estos mensajes de error personalizados en el método **messages()**:

```
public function messages()
{
    return [
        'name.required' => 'El nombre del video es requerido',
        'name.max' => 'Tamaño máximo para el nombre del video es de 100 caracteres',
        'route.mimes' => 'El formato de video debe ser .mp4 o .mpeg o .webm',
        'route.max' => 'Tamaño maximo de video 100 MB',
        'route.required' => 'El campo es requerido',
        'order.required' => 'El campo orden es requerido',
        'point_of_interest_id.required' => 'Debe seleccionar un punto de interés',
        'thematic_area_id' => 'Debe seleccionar un área temática'
    ];
}
}
```

UpdateVideoItemsRequest

Para concluir con esta parte, hemos creado una clase para la validación de las características de vídeo llamada **UpdateVideoItemsRequest**. Las reglas de validación son las siguientes:

- La calidad, el formato, la orientación y el idioma son obligatorios.
- Tamaño máximo para los campos:
    - Calidad: 100 caracteres.
    - Formato: 45 caracteres.
    - Orientación: 100 caracteres.
    - Idioma: 5 caracteres.

```
<?php

namespace App\Http\Requests;

use Illuminate\Foundation\Http\FormRequest;

class UpdateVideoItemsRequest extends FormRequest
{
  public function authorize()
  {
    return true;
  }

  public function rules()
  {
    return [
      'quality' => 'required | max: 100',
      'format' => 'required | max: 45',
      'orientation' => 'required | max: 100',
      'language' => 'required | max: 5'
    ];
  }
```

Si editamos las características de vídeo y no cumplimos con alguna de las reglas de validación, se mostrarán estos mensajes de error definidos en el método **messages()**:

```
  public function messages()
  {
    return [
      'quality.required' => 'El campo calidad debe ser requerido',
      'quality.max' => 'Tamaño maximo 100 caracteres',
      'format.required' => 'El campo formato debe ser requerido',
      'format.max' => 'Tamaño maximo 45 caracteres',
      'orientation.required' => 'El campo orientacion debe ser requerido',
      'orientation.max' => 'Tamaño maximo 100 caracteres',
      'language.required' => 'El campo lenguaje debe ser requerido',
      'language.max' => 'Tamaño maximo 5 caracteres',
    ];
  }
}
```

Proveedores

AuthServiceProvider

Con la clase proveedor **AuthServiceProvider**, hemos añadido todas las políticas que hemos creado y desarrollado en la aplicación con el fin de garantizar la seguridad. Otra forma de ignorar las políticas y tener el control absoluto de la aplicación es si un usuario es superadministrador.

```
<?php

namespace App\Providers;

use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Gate;

class AuthServiceProvider extends ServiceProvider
{
  protected $policies = [
    'App\User' => 'App\Policies\UserPolicy',
    'App\Photography' => 'App\Policies\PhotographyPolicy',
    'App\Place' => 'App\Policies\PlacePolicy',
    'App\PointOfInterest' => 'App\Policies\PointOfInterestPolicy',
    'App\ThematicArea' => 'App\Policies\ThematicAreaPolicy',
    'App\Video' => 'App\Policies\VideoPolicy',
    'Spatie\Permission\Models\Role' => 'App\Policies\RolePolicy'
  ];

  public function boot()
  {
    $this->registerPolicies();

    Gate::before(function ($user, $ability) {
      return $user->hasRole('Super Admin') ? true : null;
    });
  }
}
```

EventServiceProvider

Antes de ver el apartado siguiente, los **Eventos** de la aplicación, hemos registrado los eventos con sus escuchadores en la clase **EventServiceProvider**.

```php
<?php

namespace App\Providers;

use Illuminate\Auth\Events\Registered;
use Illuminate\Auth\Listeners\SendEmailVerificationNotification;
use Illuminate\Foundation\Support\Providers\EventServiceProvider as ServiceProvider;
use Illuminate\Support\Facades\Event;

class EventServiceProvider extends ServiceProvider
{
    protected $listen = [
        Registered::class => [
            SendEmailVerificationNotification::class,
        ],
        'App\Events\AdminInformed' => [
            'App\Listeners\UserRegistered',
        ],
        'App\Events\UserWasRegisted' => [
            'App\Listeners\SendLoginCredentials',
        ],
        'Illuminate\Auth\Events\Login' => [
            'App\Listeners\SuccessfulLogin',
        ]
    ];

    public function boot()
    {
        parent::boot();
    }
}
```

CanvasServiceProvider

Para el desarrollo de las gráficas en la parte de estadística en el dashboard, hemos creado un nuevo proveedor llamado **CanvasServiceProvider**. Proporciona un crontab ya que permite actualizar cada semana, es decir, los lunes a las 8 de la mañana los datos que han sido registrados o actualizados durante los últimos 7 días con el fin de que las gráficas vayan actualizando semanalmente.

```php
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Console\Scheduling\Schedule;

class CanvasServiceProvider extends ServiceProvider
{
  public function boot()
  {
    $this->app->booted(function () {
      $schedule = resolve(Schedule::class);
      $schedule->command('canvas:digest')
        ->weekly()
        ->mondays()
        ->timezone(config('app.timezone'))
        ->at('08:00')
        ->when(function () {
          return config('canvas.mail.enabled');
        });
    });
  }
}
```

Eventos

Recordemos que para crear un evento, primero lo añadimos en **EventServiceProvider** con uno o varios escuchadores. Después, creamos esas clases ejecutando el comando **php artisan event:generate**.

Envío de correo cuando un usuario es registrado

Events\UserWasRegisted

Cuando registramos un usuario, se ejecutará un evento que consiste en enviar un correo a ese usuario que se acaba de registrar con el fin de mostrar sus datos y una contraseña por defecto. El evento cuya clase es **UserWasRegisted** donde recibiremos como atributos un usuario y una contraseña, definidos en el constructor.

```php
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

class UserWasRegisted
{
  use Dispatchable, InteractsWithSockets, SerializesModels;

  public $user;
  public $password;

  public function __construct($user, $password)
  {
    $this->user = $user;
    $this->password = $password;
  }

  public function broadcastOn()
  {
    return new PrivateChannel('channel-name');
  }
}
```

Listeners\SendLoginCredentials

El escuchador que acompaña al evento **UserWasRegisted** es la clase **SendLoginCredentials** ya que aplicamos ese evento enviando un correo con el usuario y la contraseña generando una nueva instancia Mail. Necesitamos una clase de tipo Mail para el envío de correo.

```php
<?php

namespace App\Listeners;

use App\Events\UserWasRegisted;
use App\Mail\LoginCredentials;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Support\Facades\Mail;

class SendLoginCredentials
{
  public function handle(UserWasRegisted $event)
  {
    Mail::to($event->user)->queue(
      new LoginCredentials($event->user, $event->password)
    );
  }
}
```

Mail\LoginCredentials

En **Mail**, creamos una clase llamada **LoginCredentials** que se encarga de generar y enviar un correo recibiendo, de nuevo, como atributos un usuario y una contraseña, definidos en el constructor.

```php
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class LoginCredentials extends Mailable
{
  use Queueable, SerializesModels;

  public $user;
  public $password;

  public function __construct($user, $password)
  {
    $this->user = $user;
    $this->password = $password;
  }

  public function build()
  {
    return $this->markdown('emails.login-credentials');
  }
}
```

views\emails\login-credentials.blade.php

Contamos con la vista **email/login-credentials.blade.php** para el diseño del correo mostrando los datos recibidos del usuario con su contraseña por defecto.

```
@component('mail::message')
  <h1>Bienvenido {{ $user->name }}</h1>
  <h2>Te has registrado en nuestra aplicacion: {{ config('app.name') }}</h2>
  <p>Tus datos son:</p>
  <li>Nombre usuario: {{ $user->login }}</li>
  <li>Correo: {{ $user->email }}</li>

  Tus credenciales son: {{ $password }}

  Muchas gracias por tu colaboracion y por formar parte en este proyecto

@component('mail::button', ['url' => route('login')])
Iniciar sesion
@endcomponent

@endcomponent
```

Notificar a los administradores el registro de un usuario

Events\AdminInformed

Cuando alguien se encarga de registrar un usuario, sea un administrador o no, podemos generar un evento que permite notificar a todos los administradores que se ha registrado un nuevo usuario en la aplicación. El procedimiento es similar que el anterior.
Con la clase **AdminInformed**, contaremos con dos atributos definidos, además, en el constructor:

- **admins →** los correos de los administradores localizados desde la base de datos.
- **user →** el usuario que ha sido registrado.

```
<?php

namespace App\Events;

use Illuminate\Broadcasting\Channel;
use Illuminate\Broadcasting\InteractsWithSockets;
use Illuminate\Broadcasting\PresenceChannel;
use Illuminate\Broadcasting\PrivateChannel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;
use Illuminate\Foundation\Events\Dispatchable;
use Illuminate\Queue\SerializesModels;

class AdminInformed
{
  use Dispatchable, InteractsWithSockets, SerializesModels;

  public $admins;
  public $user;

  public function __construct($admins, $user)
  {
    $this->admins = $admins;
    $this->user = $user;
  }

  public function broadcastOn()
  {
    return new PrivateChannel('channel-name');
  }
}
```

Listeners\UserRegistered

Este escuchador **UserRegistered** recibe, desde el evento, los correos de los administradores y el usuario registrado. Lo notificamos por medio de un correo. Por tanto, contamos con una clase de

tipo **Mail**. Como son varios correos que enviar, se realiza dentro de un bucle **foreach** para cada correo a enviar.

```php
<?php

namespace App\Listeners;

use App\Events\AdminInformed;
use App\Mail\NewUser;
use Illuminate\Support\Facades\Mail;

class UserRegistered
{
  public function handle(AdminInformed $event)
  {
    foreach ($event->admins as $admin) {
      Mail::to($admin)->queue(
        new NewUser($admin, $event->user)
      );
    }
  }
}
```

Mail\NewUser

Generamos una clase **Mail** llamada **NewUser** donde recibimos como atributos los correos de los administradores y el usuario registrado, ya que ambos tienen que estar definidos en el constructor.

```php
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class NewUser extends Mailable
{
  use Queueable, SerializesModels;

  public $admins;
  public $user;

  public function __construct($admins, $user)
  {
    $this->admins = $admins;
    $this->user = $user;
  }

  public function build()
  {
    return $this->markdown('emails.new-user');
  }
}
```

views\emails\new-user.blade.php

Diseñamos y definimos el contenido de la vista **emails.new-user.blade.php**, donde mostramos el correo con los datos de ese usuario que se acaba de dar de alta en la aplicación. Dejamos preparado un enlace por si algún administrador quiere editar los datos de ese usuario nuevo.

```
@component('mail::message')
  # Nuevo usuario

  Se ha registrado un nuevo usuario en nuestro sistema: {{ $user->name . ' ' . $user->surnames }}
  Su nombre de usuario es: {{ $user->login }}
  Su correo electronico es: {{ $user->email }}

  @component('mail::button', ['url' => route('admin.users.edit', $user)])
    Editar
  @endcomponent
@endcomponent
```

Registro fecha último inicio de sesión

Listeners\SuccessfulLogin

Cuando un usuario inicie sesión, contaremos con otro evento con su escuchador. En este caso, el evento viene de **Illuminate\Auth\Events\Login** de la librería **laravel/ui**. Su escuchador asociado es la clase **SuccessfulLogin** donde, simplemente, llamara a un método declarado en el modelo **User** donde guardamos la fecha de inicio de sesión en la base de datos.

```
<?php

namespace App\Listeners;

use App\User;
use DateTime;
use Illuminate\Auth\Events\Login;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Support\Facades\Date;

class SuccessfulLogin
{
  public function handle(Login $event)
  {
    User::registerLastLogin($event);
  }
}
```

Notificación como mensaje de correo

Notifications\MyResetPassword

En este caso es una notificación, pero se podría denominar también como un evento. La clase **MyResetPassword** es una notificación en forma de mensaje de correo que será mostrado una vez introducido el correo de reseteo de contraseña. El contenido cuenta con un asunto, un mensaje, una acción y un saludo.

```php
<?php

namespace App\Notifications;

use Illuminate\Auth\Notifications\ResetPassword;
use Illuminate\Notifications\Messages\MailMessage;

class MyResetPassword extends ResetPassword
{
  public function toMail($notifiable)
  {
    return (new MailMessage)
      ->subject('Notificacion de restablecimiento de contraseña')
      ->greeting('Hola')
      ->line('Estás recibiendo este correo porque hiciste una solicitud de recuperación de contraseña para tu cuenta.')
      ->action('Recuperar contraseña', route('password.reset', $this->token))
      ->line('Si no realizaste esta solicitud, no se requiere realizar ninguna otra acción.')
      ->salutation('Saludos, '. config('app.name'));
  }
}
```

Caducidad del usuario

Una de las partes interesantes del panel de control es la caducidad de un usuario. El proceso consiste en que a través de una consola **crontab**, se realice una tarea que compruebe cada semana si el usuario se encuentra activo o inactivo. Normalmente, se centra más cuando un usuario está inactivo ya que si ese usuario lleva 3 meses sin loguearse en la aplicación, automáticamente quedará desactivado y no podrá entrar al panel de control. Para ello, hemos desarrollado lo siguiente.

Console\Kernel

Definimos una consola virtual **crontab** en la clase **Kernel** donde indicamos un nombre de comando a una tarea que se irá ejecutando semanalmente o por consola.

```php
<?php

namespace App\Console;

use Illuminate\Console\Scheduling\Schedule;
use Illuminate\Foundation\Console\Kernel as ConsoleKernel;

class Kernel extends ConsoleKernel
{
  protected function schedule(Schedule $schedule)
  {
    $schedule->command('inactive:users')->weekly();
  }
  ...
}
```

Console\Commands\CheckInactiveUsers

A continuación, hemos creado una clase de tipo **Command** llamada **CheckInactiveUsers**. En esa clase, definimos un nombre y una descripción de una tarea. En esa tarea, llama a un método del modelo **User** donde comprueba la última fecha de inicio de sesión del usuario. Si lleva 3 meses sin loguearse, entonces el usuario pasará automáticamente a inactivo guardando su valor en la base de datos.

```php
<?php

namespace App\Console\Commands;

use App\User;
use Carbon\Carbon;
use Illuminate\Console\Command;

class CheckInactiveUsers extends Command
{
  protected $signature = 'inactive:users';

  protected $description = 'Desactiva usuarios inactivos';

  public function __construct()
  {
    parent::__construct();
  }

  public function handle()
  {
    $users= User::inactiveUsers()->get();

    foreach ($users as $user){
      $this->info($user->login);
       $user->active=false;
       $user->save();
    }
  }
}
```

Para llevar a cabo este proceso, en el modelo **User**, tenemos definido un método que consulta con la base de datos comprobando si ese usuario se ha autenticado durante los últimos 3 meses.

```php
  public static function InactiveUsers()
  {
    $date = Carbon::now();
    return User::query()->where([
      ['last_login', '<', $date->subMonth(3)],
      ['active', '=', true]
    ]);
  }
```

Como ejemplo, si ejecutamos el comando **php artisan inactive:users**, encuentra uno o varios usuarios que llevaba más de 3 meses sin iniciar sesión. Por eso, el usuario pasa a ser inactivo.

```
C:\xampp\htdocs\gtv>php artisan inactive:users
pepe123

C:\xampp\htdocs\gtv>
```

→

last_login	active
2020-02-14	0

Configuración Storage

Config\filesystems.php

Para configurar la ruta de la carpeta pública de Storage, donde almacenamos imágenes o fotos y vídeos, podemos editarlo en el fichero **config\filesystem.php**.

```php
<?php

return [

  'default' => env('FILESYSTEM_DRIVER', 'local'),

  'cloud' => env('FILESYSTEM_CLOUD', 's3'),

  'disks' => [

    'local' => [
      'driver' => 'local',
      'root' => storage_path('app'),
    ],

    'public' => [
      'driver' => 'local',
      'root' => storage_path('app/public'),
      'url' => env('APP_URL').'/storage',
      'visibility' => 'public',
    ],

    's3' => [
      'driver' => 's3',
      'key' => env('AWS_ACCESS_KEY_ID'),
      'secret' => env('AWS_SECRET_ACCESS_KEY'),
      'region' => env('AWS_DEFAULT_REGION'),
      'bucket' => env('AWS_BUCKET'),
      'url' => env('AWS_URL'),
    ],

  ],

];
```

Frontend

Ahora, vamos a pasar al desarrollo en el lado del cliente, es decir, el frontend ya que se basa en el diseño del panel de control y desarrollo de eventos a la hora de realizar una acción. Contamos con una plantilla, con sus componentes y páginas personalizadas a medida que vamos desarrollando todo el contenido del panel de administración.
Para no alargar mucho este apartado, solo vamos a explicar aquel código más destacado e interesante del frontend.

Croppie.js

Esta librería de JQuery proporciona un plugin con el fin de que a la hora de pinchar una imagen aparece una ventana modal donde nos muestra un formulario con un campo de tipo imagen. Una vez subida la imagen, ajustamos el tamaño de esa imagen y lo enviamos al servidor con la imagen recortada en forma redondeada. Para llevar a cabo este proceso, hemos desarrollado lo siguiente.

users\photos.blade.php

En la vista **users\photos.blade.php**, lo primero que hemos añadido es un bloque **div** de html con los estilos de bootstrap para diseñar una ventana modal. En esa ventana modal, tenemos un campo de tipo fichero donde obtenemos la imagen, subida por Ajax, ya que vamos a reajustar el tamaño de la imagen. Tenemos preparado dos botones: el primero para enviar de nuevo la imagen recortada, y el segundo, por si queremos cancelar la operación.

```
<div class="modal " id="editPhoto" data-backdrop="static" tabindex="-1" role="dialog"
  aria-labelledby="staticBackdropLabel" aria-hidden="true">
    @csrf
    <div class="modal-dialog">
      <div class="modal-content">
        <div class="modal-header">
          <button type="button" class="close" data-dismiss="modal" aria-label="Close"><span
              aria-hidden="true">&times;</span></button>
        </div>
        <div class="modal-body">
          <img id="photo-img" class="demo-cropper cr-original-image"
            src="{{ $user->profile ? Storage::url($user->profile) : '/admin/img/default-avatar.png' }}">
          <input type="file" name="upload_image" id="upload_image" />
        </div>
        <div class="modal-footer">
          <button type="submit" class="btn btn-secondary" data-dismiss="modal">Close</button>
          <button type="submit" id="cropImageBtn" class="btn btn-primary">Recortar foto</button>
        </div>
      </div>
    </div>
</div>
```

Una de las configuraciones que tiene croppie es el zoom. En este caso, obteniendo el elemento donde recibe la imagen, definimos el tamaño de inicio de la zona de corte, que tendrá una forma redondeada. Además, ajustamos el tamaño o perímetro de toda la imagen a mostrar para llevar a cabo el reajuste de la imagen.

```
@push('scripts')
  <script>
    $(function () {
       $basic = $('#photo-img').croppie({
         destroy:true,
         viewport: {
            width: 200,
            height: 200 ,
            type:'circle'
         },
         boundary:{
          width:400,
          height:400
         },
         enableExif:true,
         enforceBoundary:true,
       });
```

Con esto, lee y obtiene la imagen a través de una ventana modal con JQuery. En esa ventana modal, editaremos una imagen que será enviada por Ajax para, más adelante, ajustarla.

```
    $('#upload_image').on('change', function(){
      var reader = new FileReader();
            reader.onload = function (event) {
              $basic.croppie('bind', {
              url: event.target.result
              }).then(function(){
                 console.log('jQuery bind complete');
              });
            }
            reader.readAsDataURL(this.files[0]);
            $('#uploadimageModal').modal('show');
      });
    });
```

El paso más importante que tenemos desarrollado es el siguiente. Obteniendo un elemento, en este caso una imagen, aplicamos un evento ya que cuando pinchemos esa imagen, carga la librería Croppie y sube una imagen por Ajax. Luego, definimos una ruta donde se lleva a cabo la edición, en este caso, de una foto de perfil. La enviamos por POST, como tipo de dato JSON. Si todo ha ido bien, esa foto se envia al servidor donde la valida y lo almacena tanto en la base de datos como en la carpeta Storage. En caso contrario, encontraremos errores en la consola del navegador.

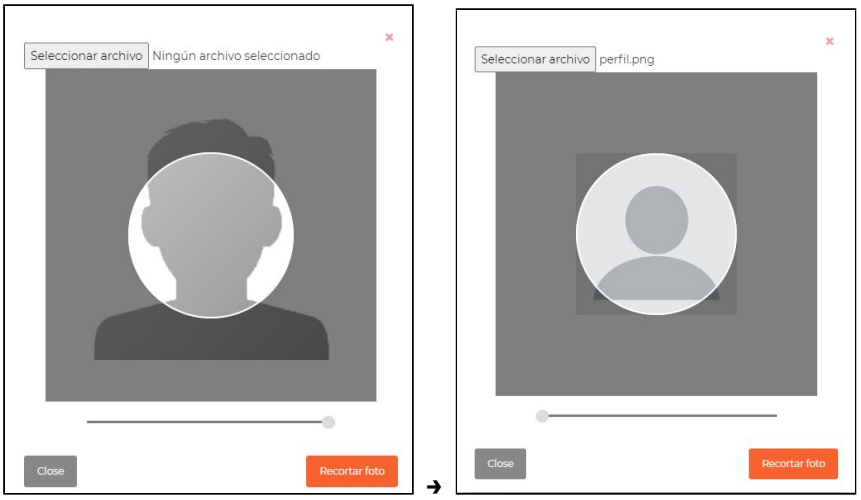
```
$('#cropImageBtn').click(function(event){
  $basic.croppie('result', {
  type: 'canvas',
  size: 'viewport'
  }).then(function(response){

  $.ajax({
   url:"/admin/users/photo/update",
    dataType: "json",
    type: "POST",
    headers: {
         'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')
      },
   data:{'profile':response,'user':  "{{$idEncript}}" },
   success:function(data) {
     window.location.hash = '#';
     location.reload();
    console.log(Object.values(data));
   },
    error:function (jqXHR, textStatus, errorThrown,XMLHttpRequest) {
    }
    }).fail(function (jqXHR, textStatus, errorThrown) {
       console.log(errorThrown);
    });
  })
});
```

users\show.blade.php

Todo este proceso se realizará, por ejemplo, en la vista **users\show.blade.php** donde en un enlace, cargaremos esa ventana modal para editar una foto de perfil.

```
...
<a href="#" data-toggle="modal" data-target="#editPhoto">
  <img class="avatar border-gray" src="{{ $user->profile ? Storage::url($user->profile) :
'/admin/img/default-avatar.png' }}">
</a>
...
```

Así queda como resultado:



Sweetalert2

La librería **Sweetalert** es otra de las librerías que hemos incorporado al panel de control. El proceso es muy sencillo, cuando pinchemos como evento un botón, mostramos un mensaje de aviso o de alerta.

layouts\app.blade.php

Si queremos cerrar sesión, como ejemplo, pero después nos arrepentimos, podemos desarrollar lo siguiente. Recibiendo un botón como elemento, aplicamos el evento **click** y, a partir de ahí, configuramos el contenido de la ventana modal como alerta:

- Especificamos un nombre y texto.
- El tipo va a ser warning
- Habilitamos el botón 'cancelar' por si queremos parar la operación.
- Definimos el texto con sus estilos para los botones.

Por último, aplicamos el método **submit()** al elemento del padre si finalmente quiere realizar la acción.
Para llevar a cabo la acción, tenemos que tener incorporada la librería.

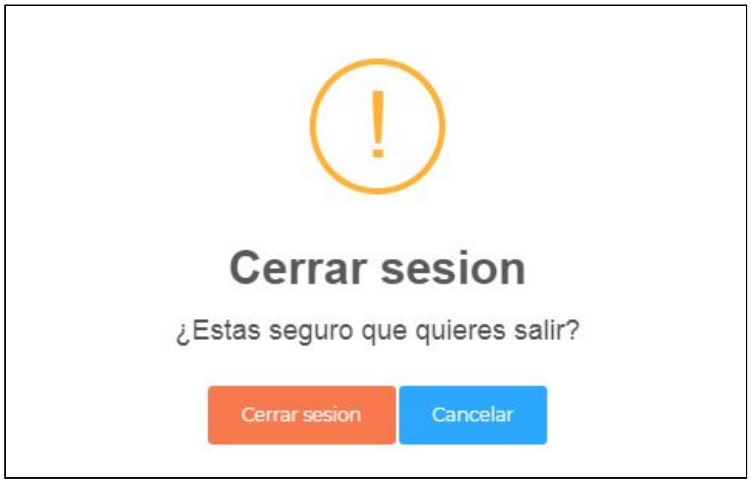
```
<script src="/admin/js/now-ui-dashboard.min.js?v=1.2.0" type="text/javascript"></script>
<script src="/admin/js/plugins/sweetalert2.min.js"></script>
<script>
  $(document).ready(function () {
    let dtable = $('#logout');
    dtable.on('click', function (e) {
      e.preventDefault();
      swal({
        title: 'Cerrar sesion',
        text: "¿Estas seguro que quieres salir?",
        type: 'warning',
        showCancelButton: true,
        confirmButtonText: 'Cerrar sesion',
        cancelButtonText: 'Cancelar',
        confirmButtonClass: 'btn btn-primary',
        cancelButtonClass: 'btn btn-info',
        buttonsStyling: false
      }).then(function () {
        e.currentTarget.parentElement.submit();
      });
    });
  });
</script>
@stack('scripts')
```

partials\nav.blade.php

En el botón de cierre de sesión que se encuentra en la vista **partials\nav.blade.php**, tenemos definido un identificador para ejecutar la ventana modal a traves de **Sweetalert2**.

```
...
<div class="dropdown-menu dropdown-menu-right" aria-labelledby="dropdown">
  <a  href="{{route('admin.users.show', auth()->user()->id)}}" class="dropdown-item text-dark">Mi perfil</a>
  <form action="{{ route('logout') }}" method="POST">
    @csrf
    <button type="submit" class="dropdown-item" id="logout">Cerrar sesión</button>
  </form>
</div>
...
```

Este es el resultado:



users\index.blade.php

Un detalle para concluir. Si borramos un dato y nos muestra una alerta pero no queremos borrar, al pinchar 'Cancelar' se genera otra alerta mostrando un nuevo mensaje. Para ello, en la parte final de la configuración de sweetalert, definimos una condición comprobando si ha pinchado ese botón.

```
...
, function (dismiss) {
  if (dismiss === 'cancel') {
    swal(
      'Cancelado la operacion',
    )
  }
});
...
```

Este es el resultado:

Simple-qr

Para el diseño de un qr, la librería **simple-qr** simplemente genera un icono qr que cuenta con un método donde definimos el tamaño y qué valor queremos que se genere cuando estemos escaneando. Puede ser un valor, un numero de telefono, una url, etc.

pointsofinterest\show.blade.php

Tanto en las vistas **pointsofinterest\show.blade.php** como **pointsofinterest\edit.blade.php** hemos incorporado este método con el fin de generar el icono de qr: **QrCode::size(tamaño)->generate(valor')**.

```
...
<div class="text-center">
  <span class="font-weight-bold">Escanea el codigo qr a mostrar el punto de interés desde tu
movil</span>
  {!!QrCode::size(300)->generate( 'https://goo.gl/maps/' . $pointsofinterest->qr ) !!}
  <p><span class="font-weight-bold">Qr:</span> {{ $pointsofinterest->qr }}</p>
</div>
...
```

Con esto generamos y mostramos un qr. Este es el resultado:

Estadística y Laravel-charts

Esta parte corresponde al diseño de las estadísticas. Para ello, hemos incorporado la librería **Laravel-charts** que cuenta con una serie de métodos para la creación y diseño de las gráficas definidos desde el controlador. Hemos desarrollado unas vistas con el fin de mostrar ese resultado.

statistics\chart.blade.php

En la primera vista **statistics\chart.blade.php**, en una card, diseñamos el contenido y los estilos de las gráficas. Desde el controlador, aplicamos el modelo que representa, por un lado, los datos que van actualizando semanalmente y, por otro, el script a la hora de actualizar las gráficas. Con esto, obtendremos el diseño de las gráficas de visitas y puntos de interés creados.

```
<div class="card card-chart">
  <div class="card-header">
    <h4 class="card-title"><font style="vertical-align: inherit;"><font style="vertical-align: inherit;">{{$title}}</font></font></h4>
  </div>
  <div class="card-body">
    <div class="chart-area">
      <div class="chartjs-size-monitor" style="position: absolute; left: 0px; top: 0px; right: 0px; bottom: 0px; overflow: hidden;
pointer-events: none; visibility: hidden; z-index: -1;">
        <div class="chartjs-size-monitor-expand"
style="position:absolute;left:0;top:0;right:0;bottom:0;overflow:hidden;pointer-events:none;visibility:hidden;z-index:-1;">
          <div style="position:absolute;width:1000000px;height:1000000px;left:0;top:0"></div>
        </div>
      <div class="chartjs-size-monitor-shrink"
style="position:absolute;left:0;top:0;right:0;bottom:0;overflow:hidden;pointer-events:none;visibility:hidden;z-index:-1;">
        <div  style="position:absolute;width:200%;height:200%;left:0; top:0"></div>
      </div>
    </div>
    {!! $model->container() !!}
  </div>
</div>
<div class="card-footer">
  <div class="stats">
    <i class="now-ui-icons ui-2_time-alarm"></i>
      <font style="vertical-align: inherit;">
          <font style="vertical-align: inherit;"> Los últimos 7 días</font>
      </font></div>
  </div>
</div>
@push('scripts')
  {!! $model->script() !!}
@endpush
```

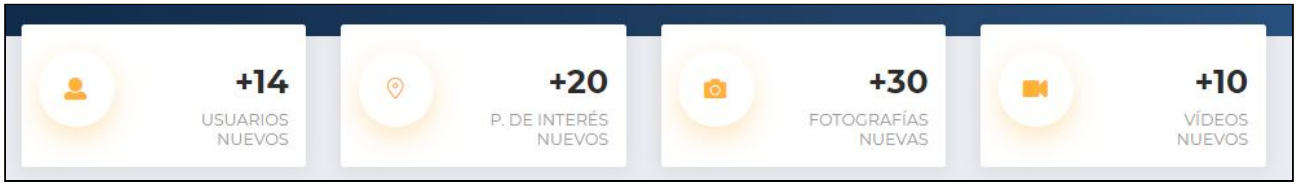
Así quedan las gráficas diseñadas:

statistics\newThings.blade.php

En la vista **statistics\newThings.blade.php** donde hemos diseñado bloques en forma de 'card' para la contabilidad de nuevos usuarios, puntos de interés, fotos y vídeos.

```
<div class="col-lg-3 col-sm-6">
  <div class="card card-stats">
    <div class="card-body ">
      <div class="statistics statistics-horizontal">
        <div class="info info-horizontal">
          <div class="row">
            <div class="col-5">
              <div
                class="icon icon-warning icon-circle">
                <i class="{{$icon}}"></i>
              </div>
            </div>
            <div class="col-7 text-right">
              <h3 class="info-title">
                +{{$number}}</h3>
              <h6 class="stats-title">{{$title}}</h6>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>
```

Este es el resultado:

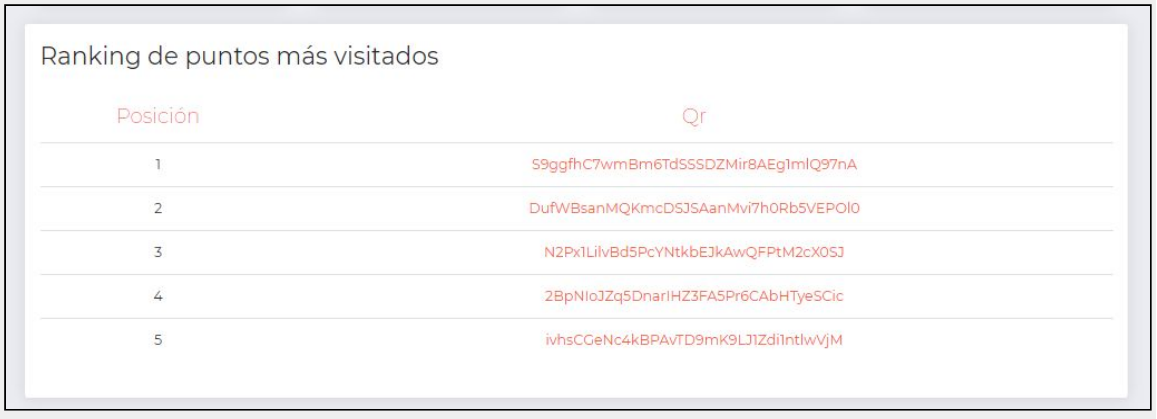


statistics\ranking.blade.php

Para el ranking de los puntos de interés más visitados, contamos con la vista **statistics\ranking.blade.php** donde, en una tabla, vamos a listar por posiciones, esos puntos de interés más visitados recibidos por el controlador.

```
<div class="card">
  <div class="card-header">
    <span class="card-title h4">Ranking de puntos más visitados</span>
  </div>
  <div class="card-body">
    <div class="table-responsive">
      <table id="users-table" class="table text-center">
        <thead class="text-primary">
        <th class="text-center">Posición</th>
        <th>Qr</th>
        </thead>
        <tbody>
        @foreach($pointsOfInterestMostVisits as $pointOfInterest)
          <tr>
            <td class="text-center">{{ $loop->index+1}}</td>
            <td>
              <a href="{{ route('admin.pointsofinterest.show',$pointOfInterest) }}">{{ $pointOfInterest->qr }}</a>
            </td>
          </tr>
        @endforeach
        </tbody>
      </table>
    </div>
  </div>
</div>
```

Este es el resultado del diseño de un ranking:



Jasny-bootstrap

Además, hemos incorporado una librería de bootstrap llamada **Jasny-bootstrap** que consiste en dar estilos al campo de tipo fichero o imagen gracias a las clases que proporciona esta librería.

photographies\edit.blade.php

Esta librería la hemos utilizado en la vista **photographies\edit.blade.php** donde tenemos un campo de tipo imagen para subir una foto. Con la clase **fileinput**, aplicamos sus estilos y cuenta con un grupo de botones:

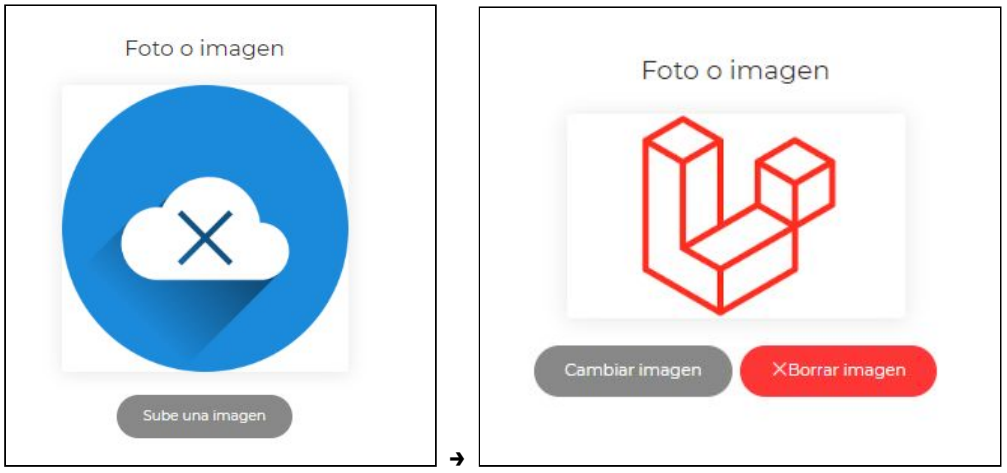
- Un primer botón de inicialización donde subiremos una imagen.
- Cuando hayamos subido una imagen en el campo, tendremos preparados dos botones
    - Un primer botón que cambie una imagen cuando la clase indique que se ha subido una imagen previa.
    - Un segundo botón de tipo enlace que borre la imagen, siempre que la clase indique que hay imagen subida.

```
...
<div class="fileinput fileinput-new text-center" data-provides="fileinput">
  <div class="fileinput-new thumbnail img-raised">
    <img src="{{ old('route', $photography->route ? Storage::url($photography->route) : '/admin/img/notAvailable.png') }}">
  </div>
  <div class="fileinput-preview fileinput-exists thumbnail img-raised"></div>
  <div>
  <span class="btn btn-raised btn-round btn-default btn-file">
    <span class="fileinput-new">Sube una imagen</span>
    <span class="fileinput-exists">Cambiar imagen</span>
    <input type="file" name="route">
  </span>
    <a href="#" class="btn btn-danger btn-round fileinput-exists" data-dismiss="fileinput"><i class="now-ui-icons
ui-1_simple-remove"></i>Borrar imagen</a>
  </div>
</div>
...
```

Para hacer efecto a los estilos del campo, tenemos que tener incorporado el fichero **jasny-bootstrap.min.js**.

```
@push('scripts')
  <script src="/admin/js/plugins/jasny-bootstrap.min.js"></script>
...
```

Así queda el resultado:



Datatables

Para terminar, hemos utilizado la librería **datatables.js** de jquery donde crea y diseña tablas dinámicas y con grandes estilos. Gracias a esta librería, podemos ordenar columnas, buscar un dato en concreto, tener una paginación, etc.

Normalmente, esta librería se encuentra incorporada y usada en todas las vistas **index.blade.php**. En la configuración de la librería, obtenemos el elemento id de una tabla. Utilizando la librería, configuramos como queremos que se muestre la tabla y sus estilos. Además, obtenemos el elemento id de un buscador para dar funcionalidad a la búsqueda de un dato concreto. Con el evento **keyup**, actualizaremos la tabla hasta encontrar ese dato coincidiendo con el carácter introducido y recibido en un campo del formulario.

```
...
<script type="text/javascript" src="/admin/js/datatables.min.js"></script>
<script>
  $(function () {
    var table =   $('#users-table').DataTable({
      "paging": true,
      "lengthChange": false,
      "ordering": true,
      "info": true,
      "autoWidth": false,
    });
    $('#search').on( 'keyup', function () {
          table.search( this.value ).draw();
        } );
    $('#users-table_filter').hide();
  });
</script>
...
```

Para completar el proceso, vamos a crear y diseñar un buscador encima de una tabla con sus identificadores para dar funcionalidad a la librería.

```
...
<div class="card-body">
  <form class="form-inline d-flex justify-content-center md-form form-sm mt-0">
    <i class="fas fa-search" aria-hidden="true"></i>
    <input class="form-control form-control-sm ml-3 w-75" aria-controls="users-table" id="search" type="text"
placeholder="Search"
      aria-label="Search">
  </form>
  <div class="table-responsive">
    <table id='users-table' class="table">
      <thead class="text-primary">
      <th class="text-center">Foto de perfil</th>
      <th class="text-center">Id</th>
      <th>Nombre Usuario</th>
      <th>Correo</th>
      <th>Nombre</th>
      <th>Apellidos</th>
      <th>Area tematica</th>
      <th>Rol</th>
      <th>Activo</th>
      <th>Acciones</th>
      </thead>
      <tbody>
        ...
      </tbody>
    </table>
  </div>
</div>
...
```

Así queda el diseño de la tabla con el buscador encima:

A medida que vayamos introduciendo un carácter o nombre en el buscador, gracias al evento, conseguimos encontrar, cómodamente, un dato concreto.

Anexos

Tareas pendientes en GTV

Este es el desarrollo del proyecto GTV. Todavía quedan tareas pendientes por hacer ya que se van a desarrollar muy pronto. Las tareas son las siguientes:

- **Desarrollo de un buscador general →** Siempre es necesario un buscador para listar todos los datos a medida que introducimos un carácter o palabra.
- **Registro automático de características de vídeo →** Cuando registramos un vídeo en la base de datos, como hemos generado automáticamente una fila con el id de ese vídeo, podemos insertar automaticamentes unos valores por defecto a los campos de la tabla características de vídeo.
- **Caducidad de los administradores →** Si a un administrador se le caduca su cuenta, no solo no puede loguearse sino que podría dañar la aplicación. Para ello, tenemos que desarrollar la forma de que los administradores no caduquen sus cuentas.
- **Desarrollo en pruebas →** Es fundamental tener desarrollado un conjunto de clases para hacer tests de prueba y comprobar la funcionalidad de una parte de la aplicación.
- **Modificación o mejora de perfil del usuario →** Podemos mejorar el perfil de usuario con más datos personales, por ejemplo, o con un diseño mejorado.
- **Listado de usuarios en inicio de sesión →** Para la estadística, podemos listar, en una tabla, los usuarios que han iniciado sesión durante la última semana, por ejemplo, guardando el último inicio de sesión, sea fecha y hora o con un texto 'hace x minutos/horas/días/año(s)'.
- **Mejorar la versión del frontend →** css, bootstrap, JavaScript o JQuery. Si fuese necesario, una librería Bootstrap o JQuery.
- **laravel-collective** o **Stydenet/HTML →** Sería necesario incorporar las librerías **laravel-collective** o **Stydenet/HTML**, ya que tenemos demasiados formularios hechos en el dashboard. Gracias a eso, no solo definimos componentes para cada campo del formulario sino estaríamos reduciendo código en las vistas **create** y **edit**.
- **Desarrollo de archivos log**.
- **Mejorar las validaciones desarrolladas →** Las validaciones las tenemos desarrolladas en las clases Requests. Será necesario proporcionar más tipos de validaciones por ejemplo:
    - **Texto →** el primer carácter que sea mayúscula en los nombres y apellidos , una expresión regular, que no sea un número, admiten caracteres especiales, mayúsculas, números en la contraseña. Validar el formato de un correo, un código qr, id de dispositivo, idioma, etc.
    - **Números →** número decimal y un número máximo y mínimo en la latitud y longitud, que no admita caracteres.
    - **Archivos →** proporcionar y mejorar validaciones para la subida de vídeos y fotos.
- **Redimensionar tamaño de vídeos y fotos →** Cuando registremos un vídeo o una foto, tendremos que tener desarrollada la forma de redimensionar o modificar el tamaño del archivo.