CREACIÓN DE UNA API REST CON LARAVEL

2020 - Chalo Salvador (github.com/chalosalvador)

TABLA DE CONTENIDOS

INTRODUCCIÓN

QUÉ ES UNA API

- Aplication programming interface
- Es la interfaz que hace posible la interacción entre aplicaciones. Es la manera en la que un cliente puede interactuar con un servidor para pedir un recurso.
- Por ejemplo, imagina que estás en un restaurante y quieres pedir un plato en específico (recurso) a la cocina (servidor). No hay forma de acceder directamente y tomarlo si no que se necesita un mesero (API) para poder hacer el pedido a la cocina. En este escenario podríamos decir que el menú contiene los puntos de acceso (endpoints) hacia cada unos de los recursos.
- Un API no es la base de datos o el servidor, es el código que sirve de punto de acceso al servidor y sus recursos.

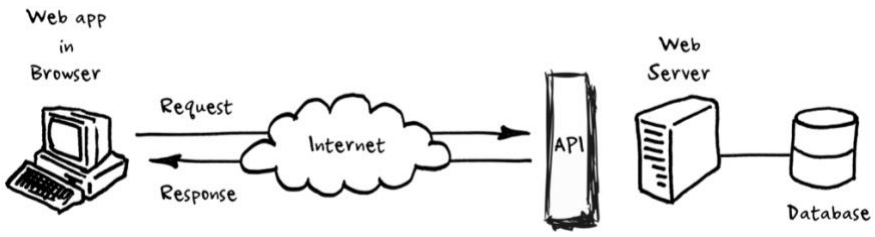

Ilustración 1 Flujo de una API

- Se puede ver una API como una aplicación típica de servidor que necesita de URL para acceder a los recursos, pero en este caso en lugar de devolver HTML, CSS y JS, los recursos son devueltos en formatos como JSON, XML, YAML, HTML.
- Una API permite escalabilidad, portabilidad esto quiere decir que podemos usar la misma lógica para construir una aplicación que pueda ser accedida desde cualquier tipo de clientes: navegador, app móvil, televisor, etc.
- Existen varios ejemplos de API en la industria de Internet:
    1. Google (https://developers.google.com/apis-explorer)
    2. Twitter (https://developer.twitter.com/en/docs/api-reference-index)
    3. Facebook (https://developers.facebook.com/docs/apis-and-sdks?locale=es_ES)
    4. Expedia (https://expediapartnersolutions.com/products/api)
    5. GitHub (https://developer.github.com/v3/)

JSON – JAVASCRIPT OBJECT NOTATION

- Es el formato más utilizado para intercambiar datos, es simple y ligero.
- JSON permite que una API sea escalable e independiente de la plataforma (portable) ya que cualquier lenguaje moderno puede leerlo.
- https://jsonlint.com/

```
{
  "user": {
          "name": "Chalo",
          "lastname": "Salvador",
          "email": "chalosalvador@gmail.com",
          "github": "https://github.com/chalosalvador"
  }
}
```

REPRESENTATIONAL STATE TRANSFER (REST)

- Término introducido en el año 2000 por Roy Fielding en su tesis doctoral (https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm)
- Es un conjunto de principios de arquitectura de software para construir aplicaciones que trabajan mediante HTTP. El principal ejemplo es la WWW.
- Una API REST permite la interacción con los recursos mediante los métodos HTTP principalmente: POST (crear), GET (consultar), PUT (editar), DELETE (eliminar). Otros métodos son HEAD, PATCH, CONNECT, TRACE, OPTIONS. Más detalles en: https://developer.mozilla.org/es/docs/Web/HTTP/Methods.
- En REST todo es un recurso accesible mediante una URI (unique resource identifier)
- Según Fielding las restricciones que definen a un sistema RESTful son:
    1. **Cliente-servidor:** esta restricción mantiene al cliente y al servidor débilmente acoplados. Esto quiere decir que el cliente no necesita conocer los detalles de implementación del servidor y el servidor se "despreocupa" de cómo son usados los datos que envía al cliente.
    2. **Sin estado:** no se mantiene un estado en la comunicación entre el cliente y el servidor. Cada petición que haga el cliente debe contener toda la información necesaria para que el servidor entienda la petición. Todas las sesiones se manejan directamente en el cliente (Cookies o localStorage). Esto permite visibilidad, fiabilidad (recuperación frente a fallos) y escalabilidad ya que se pueden aumentar más nodos o servidores y no hace falta que estos sincronicen el estado de la aplicación.
    3. **Cacheable:** debe admitir un sistema de almacenamiento en caché. La infraestructura de red debe soportar una caché de varios niveles. Este almacenamiento evitará repetir varias conexiones entre el servidor y el cliente para recuperar un mismo recurso.
    4. **Interfaz uniforme:** define una interfaz genérica para administrar cada interacción que se produzca entre el cliente y el servidor de manera uniforme, lo cual simplifica y separa la arquitectura. Esta restricción indica que cada recurso del servicio REST debe tener una única dirección, "URI".
    5. **Sistema de capas:** el servidor puede disponer de varias capas para su implementación, es decir que cada componente no puede ver más allá de las capas inmediatamente inferior o superior. Esto ayuda a mejorar la escalabilidad, el rendimiento y la seguridad.
    6. **Código a demanda:** Permite añadir funcionalidad en el cliente mediante la descarga de scripts o applets. Esto simplifica al cliente y mejora la extensibilidad. Esta restricción es opcional.
- API RESTful es el nombre que se le da a las API que cumplen con todas las restricciones que impone REST.
- Propiedades de los métodos HTTP:
    - **Método seguro**: no modifica datos en el servidor y puede ser cacheado
    - **Método idempotente**: el resultado es independiente del número de invocaciones. Ejemplo: x=2 (idempotente), x=x+1 (no idempotente)
    - **POST:** no seguro, no idempotente
    - **GET:** seguro e idempotente
    - **PUT:** no seguro, si idempotente
    - **DELETE:** no seguro, si idempotente

Método HTTP	Seguro	Idempotente
GET	Sí	Sí
POST	No	No
PUT	No	Sí
DELETE	No	Sí

- Todo recurso tiene una ruta diferente (URI)
- Las rutas suelen tomar el plural del recurso, por ejemplo:

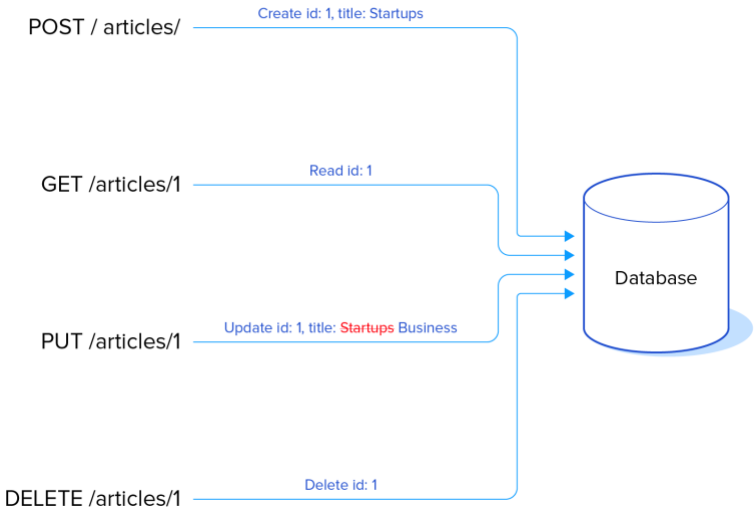

Ilustración 2 Ejemplo endpoints API

¿QUÉ ES LARAVEL?

- Laravel es un framework de PHP enfocado en mejorar la productividad de los programadores y ahorrarnos tiempo.
- El framework toma varias librerías ya creadas y las abstrae para que puedan usarse de una manera más sencilla.
- Es orientado a convenciones sobre configuraciones, es decir, impone algunas convenciones para evitar que el desarrollador tenga que realizar varias configuraciones tediosas.
- Laravel se mantiene actualizado con la web moderna incorporando características como comunicación en tiempo real o notificaciones.

INSTALACIÓN

Referencia: https://laravel.com/docs/6.x/installation

REQUISITOS

- XAMPP (PHP, MySQL): https://www.apachefriends.org/es/index.html
- IDE (PhpStorm): https://www.jetbrains.com/es-es/phpstorm/
    - Licencia educativa: https://www.jetbrains.com/es-es/community/education#students

INSTALACIÓN DE COMPOSER

Composer es el gestor de dependencias de PHP. Para instalarlo se deben seguir las instrucciones de su documentación de acuerdo con el sistema operativo: https://getcomposer.org/doc/00-intro.md

Asegurarse de colocar el directorio `bin` global de Composer en la variable de entorno `$PATH` (https://getcomposer.org/doc/00-intro.md#globally) para que el ejecutable de Laravel pueda ser localizado en el sistema. Este directorio existe en diferentes ubicaciones según el sistema operativo; sin embargo, algunas de las ubicaciones más comunes son las siguientes:

- macOS: `$HOME/.composer/vendor/bin`
- Distribuciones GNU/Linux: `$HOME/.config/composer/vendor/bin` o `$HOME/.composer/vendor/bin`
- Windows: `%USERPROFILE%\AppData\Roaming\Composer\vendor\bin`

También se puede encontrar la ruta global de instalación de Composer ejecutando `composer global about` y observando la primera línea.

INSTALACIÓN DE LARAVEL

Mediante el instalador de Laravel:

```
composer global require laravel/installer
```

CREACIÓN DE PROYECTO

Referencia: https://laravel.com/docs/6.x/installation

Para crear el proyecto de Laravel se lo puede hacer con el siguiente comando:

```
laravel new blog
```

O utilizando composer se puede especificar una version de Laravel. La version 6 es LTS por lo que es más recommendable.

```
composer create-project --prefer-dist laravel/laravel blog "6.*"
```

SERVIDOR DE DESARROLLO LOCAL

El framework viene con un servidor de desarrollo local, para levantarlo lo hacemos con el siguiente comando:

```
php artisan serve
```

ESTRUCTURA DE DIRECTORIOS

Referencia: https://laravel.com/docs/6.x/structure

Antes de empezar es importante dejar claro que Laravel omite la creación por defecto del directorio `model` debido a que el término tiene diferentes interpretaciones para diferentes desarrolladores, así que por defecto las clases correspondientes a los modelos Eloquent son ubicados directamente en el directorio `app`, y depende de cada desarrollador si decide moverlo a un directorio diferente. En este caso dejaremos los modelos en la ubicación por defecto y nos referiremos a modelos para indicar las clases que interactúan o que representan una entidad de la base de datos.

En el directorio raíz tenemos los siguientes directorios:

- **app:** El directorio `app` contiene el código principal de la aplicación.
- **bootstrap:** contiene el archivo `app.php` que maqueta el framework. Este directorio también almacena un directorio `cache` que contiene archivos generados por el framework para optimización de rendimiento como los archivos de cache de rutas y servicios.
- **config:** contiene todos los archivos de configuración de tu aplicación. Todos están muy bien documentados.
- **database:** contiene las migraciones de la base de datos, model factories y seeders.
- **public:** contiene todos los archivos de acceso público de la aplicación, como el archivo `index.php`, el cual es el punto de acceso para todas las solicitudes que llegan y configura la autocarga. Este directorio también almacena los assets, tales como imágenes, JavaScript y CSS.
- **resources:** contiene los templates de las vistas así como también los assets sin compilar tales como LESS, Sass o JavaScript. Este directorio también almacena todos los archivos de idioma.
- **routes:** contiene todas las definiciones de rutas para la aplicación. Laravel incluye algunos archivos de rutas: `web.php`, `api.php`, `console.php` y `channels.php`. Para este curso nos enfocaremos en el archivo `api.php` el cual contiene rutas que pertenecen al grupo de middleware `api`. Estas rutas están pensadas para no tener estado, así que las solicitudes que llegan a la aplicación a través de estas rutas están pensadas para ser autenticadas mediante tokens y no tendrán acceso al estado de sesión.
  Para detalles de los demás archivos de rutas se pueden referir a la documentación oficial.
- **storage:** Este directorio está segregado en los directorios `app`, `framework` y `logs`. El directorio `app` puede ser usado para almacenar cualquier archivo generado por la aplicación. El directorio `framework` es usado para almacenar archivos generados por el framework y caché. Finalmente, el directorio `logs` contiene los archivos de registros de la aplicación.
  El directorio `storage/app/public` puede ser usado para almacenar archivos generados por el usuario, tales como imágenes de perfil, que deberían ser accesibles públicamente. Se debe crear un enlace simbólico en `public/storage` que apunte a este directorio. Para crear el enlace se usa el comando `php artisan storage:link`.
- **tests:** contiene las pruebas automatizadas. Una prueba de ejemplo de PHPUnit es proporcionada. Cada clase de prueba debe terminar con `Test`. Para ejecutar las pruebas se usan los comandos `phpunit` o `php vendor/bin/phpunit`.
- **vendor:** contiene todas las dependencias instaladas por composer.

Para este proyecto se interactuará mayormente con el directorio `app/Http` donde se crearán los controladores, middlewares y requests. Para más detalles de otros directorios que se irán creando dentro de `app/Http` se puede consultar la documentación: https://laravel.com/docs/6.x/structure#the-app-directory.

CONFIGURACIÓN

Referencia: https://laravel.com/docs/6.x/configuration

ARCHIVO `CONFIG/APP.PHP`

En este archivo inicialmente se debe cambiar los valores de las siguientes opciones:

```
'timezone' => 'America/Guayaquil',

'locale' => 'es',

'faker_locale' => 'es_EC',
```

- Para ver todas las zonas horarias disponibles: https://www.php.net/manual/es/timezones.america.php

ARCHIVO `.ENV`

En este archivo se deben tomar en cuenta las siguientes configuraciones antes de empezar el proyecto:

```
APP_NAME=Blog
APP_ENV=local
APP_KEY=base64:xz22DqTN0eOql6PopUdxecLqoA27EFjK3ecmcEImbZs=
APP_DEBUG=true
APP_URL=http://localhost
```

- Base de datos:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=
```

- Servidor de correos:

```
MAIL_MAILER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
MAIL_FROM_ADDRESS=null
MAIL_FROM_NAME="${APP_NAME}"
```

- Para verificar que está activo el módulo `mod_rewrite` podemos hacerlo en el archivo `httpd.conf` en Mac con XAMPP instalado se encuentra en: `/Applications/XAMPP/etc/httpd.conf`

GIT Y GITHUB

CREACIÓN DE REPO

- Crear un nuevo repo en GitHub y luego ejecutar los siguientes comandos:

```
git remote add origin URL_REPO
git add .
git commit -m "Inicialización de proyecto"
git push origin master
```

CREACIÓN DE RAMAS

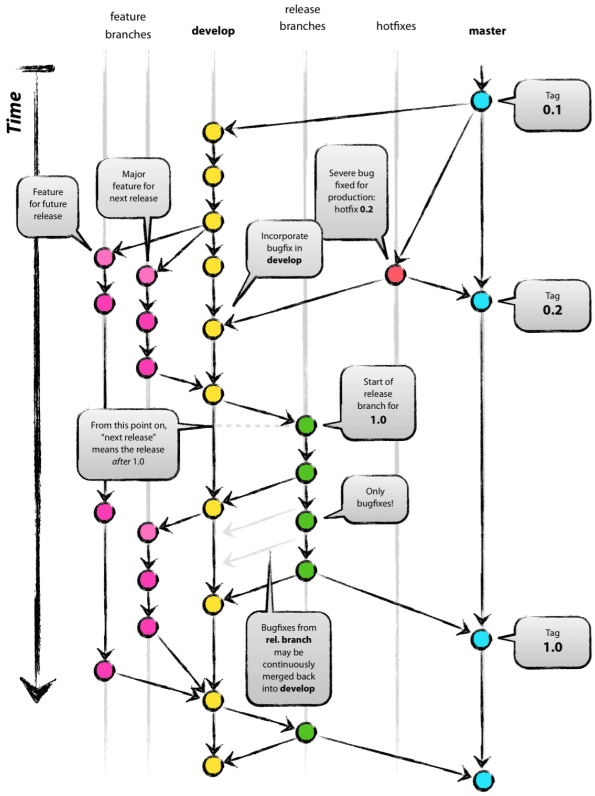

Ilustración 3 Flujo de trabajo Git

https://nvie.com/posts/a-successful-git-branching-model/

- Más material sobre Git y las ramas de Git:
  - Git la guía sencilla: https://rogerdudler.github.io/git-guide/index.es.html

- Creación de una rama local

```
git checkout -b dev
```

- Subir la rama local al repositorio remote

```
git push origin dev
```

PROTECCIÓN DE RAMAS

- Proteger las ramas `master` y `dev` para evitar que reciban push directos y obligar a la creación de ramas y pull requests.

CONFIGURACIÓN DE ZENHUB

- Instalar la extensión de Chrome Zenhub
- Crear el tablero con las columnas
    - Icebox
    - Backlog
    - Sprint backlog
    - In progress
    - Review/QA
    - Closed
- Creación de etiquetas
- Creación de tareas
- Creación de milestones
- Creación de estimaciones
- Creación de pull requests
- Merge PR

COMMITS

- Los commits deben seguir siempre el siguiente formato:

```
git commit -m "#1 registro de usuarios"
```

    - El mensaje debe empezar con # y el número de la tarea a la que pertenece este commit. Esto permite que todo commit sea relacionado a alguna tarea del tablero y se lo mostrará así en GitHub.
    - El título del commit debe ser corto.
    - Se puede añadir más detalles si se mantienen abiertas las comillas y se da enter, cuando se termine de escribir se cierran las comillas y se presiona enter.
- Para subir los cambios a GitHub ejecutamos

```
git push origin nombre-de-rama
```

    - Esto nos dará la opción de crear un pull request.

PULL REQUESTS

- Los pull request (o merge request) nos permiten tener un mejor control sobre la calidad de nuestro código cuando trabajamos en equipo. Nos permite ver los cambios y pedir correcciones o aceptarlos para unirlos al código principal.
- Al crearlos podemos asignar a alguien que se encargue de revisarlo.
- La persona asignada para revisión puede descargar la rama, probar que todo funcione bien y aceptar o pedir cambios con comentarios y capturas de pantalla.
- Todos los PR se harán por defecto a la rama dev que es nuestra rama por defecto.

PUSH TO GITHUB

INICIO DEL PROYECTO

El proyecto que vamos a desarrollar consiste en un blog donde los usuarios podrán leer artículos escritos por otros usuarios. Pueden registrarse e iniciar sesión para escribir sus propios artículos o comentar sobre los artículos de otros. Las entidades se conformarán el sistema se ven a continuación.

ENTIDADES

- articles
  - title
  - body
  - author
  - created_at
  - updated_at
- users
  - name
  - email
  - email_verified_at
  - password
  - remember_token
  - created_at
  - updated_at
- comments
  - text
  - article
  - author
  - created_at
  - updated_at

MODELO DE LA BASE DE DATOS

- Este es el modelo de la base de datos del sistema que vamos a desarrollar.

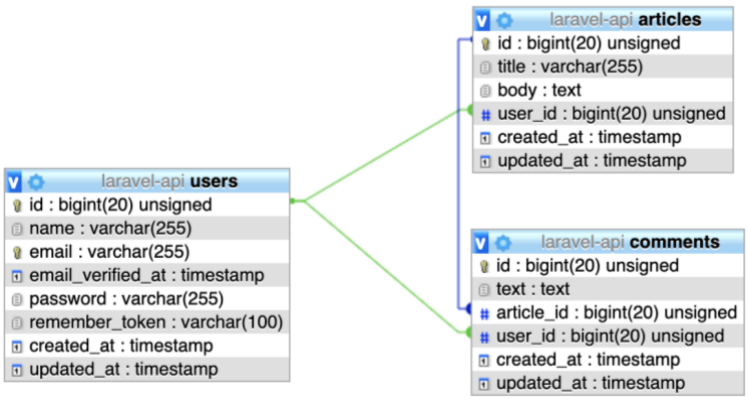

Ilustración 4 Modelo BDD

CREACIÓN DEL MODELO ARTICLE

Referencia: https://laravel.com/docs/6.x/eloquent#defining-models

El comando para crear un modelo es el siguiente:

```
php artisan make:model Article -m
```

`-m` crea la migración para el modelo (`--migration`)

MIGRACIONES

Al ejecutar el commando anterior se crea la migración para `Article` que se ve similar al siguiente código:

```php
<?php

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

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

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

- `up()` y `down()` son los métodos que corren cuando se hace la migración o el rollback de la migración
- `$table->increments('id')` configura el campo con un entero autoincrement con el nombre `id`
- `$table->timestamps()` configura los campos `created_at` y `updated_at`, que tendrán automáticamente el valor de la hora actual del servidor cuando un registro en la tabla es creado o actualizado.
- `Schema::dropIfExists()` Elimina la tabla si existe al momento de hacer la migración.

Agregar dos campos a la migración de `Article` de la siguiente manera:

```
public function up()
{
    Schema::create('articles', function (Blueprint $table) {
        $table->increments('id');
        $table->string('title');
        $table->text('body');
        $table->timestamps();
    });
}
```

- `string()` columna `VARCHAR`
- `text()` columna `TEXT`

EJECUTAR LA MIGRACIÓN

Para ejecutar la migración se utiliza el siguiente comando:

```
$ php artisan migrate
```

Laravel viene con dos migraciones por defecto:
`create_users_table` y `create_password_resets_table`

Ahora, para permitir que los campos sean llenados con datos del cliente, se debe añadir los campos a la propiedad `$fillable` del modelo `Article`.

```
class Article extends Model
{
    protected $fillable = ['title', 'body'];
}
```

Esto permite que estos campos puedan ser llenados desde los métodos `create()` y `update()` es decir, que puedan ser llenados por las peticiones mediante formularios por peticiones POST del API.

También se puede usar la propiedad `$guarded` para evitar que los campos sean llenados.

PUSH TO GITHUB

SEEDING DE LA BASE DE DATOS

Referencia: https://laravel.com/docs/6.x/seeding

Este proceso nos permite llenar la base de datos con datos ficticios para probar nuestra aplicación. Laravel nos proporciona la librería **Faker** para generar estos datos basados en nuestros modelos.

```
php artisan make:seeder ArticlesTableSeeder
```

Los seeders se ubican en el directorio `/database/seeds`

```
class ArticlesTableSeeder extends Seeder
{
    public function run()
    {
        // Vaciar la tabla.
        Article::truncate();

        $faker = \Faker\Factory::create();

        // Crear artículos ficticios en la tabla
        for ($i = 0; $i < 50; $i++) {
            Article::create([
                'title' => $faker->sentence,
                'body' => $faker->paragraph,
            ]);
        }
    }
}
```

Correr el seed:

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

Seeder para Users

```
class UsersTableSeeder extends Seeder
{
    public function run()
    {
        // Vaciar la tabla
        User::truncate();

        $faker = \Faker\Factory::create();

        // Crear la misma clave para todos los usuarios
        // conviene hacerlo antes del for para que el seeder
        // no se vuelva lento.
        $password = Hash::make('123123');

        User::create([
            'name' => 'Administrador',
            'email' => 'admin@prueba.com',
            'password' => $password,
        ]);

        // Generar algunos usuarios para nuestra aplicacion
        for ($i = 0; $i < 10; $i++) {
            User::create([
                'name' => $faker->name,
                'email' => $faker->email,
                'password' => $password,
            ]);
        }
```

```
    }
}
```

Añadir los seeder al `DatabaseSeeder` principal

```
class DatabaseSeeder extends Seeder
{
    public function run()
    {
        $this->call(ArticlesTableSeeder::class);
        $this->call(UsersTableSeeder::class);
    }
}
```

Así Podemos correr `$ php artisan db:seed` para que corran todos los seeders

PUSH TO GITHUB

RUTAS Y CONTROLADORES

CREAR LAS RUTAS DEL API

En esta sección se crearán los endpoints para nuestra aplicación, empezaremos por el modelo `Article` con las acciones crear, obtener lista, obtener uno, actualizar y eliminar. Para esto modificamos el archivo `routes/api.php` y agregamos el siguiente código:

```
Use App\Article;

Route::get('articles', function() {
    return Article::all();
});

Route::get('articles/{id}', function($id) {
    return Article::find($id);
});

Route::post('articles', function(Request $request) {
    return Article::create($request->all());
});

Route::put('articles/{id}', function(Request $request, $id) {
    $article = Article::findOrFail($id);
    $article->update($request->all());

    return $article;
});

Route::delete('articles/{id}', function($id) {
    Article::find($id)->delete();

    return 204;
});
```

Todas las rutas dentro de `api.php` tendrán el prefijo `/api/`, si queremos quitar el prefijo se lo puede hacer en el archivo: `app/Providers/RouteServiceProvider.php`

TEST ON POSTMAN

PUSH TO GITHUB

CREACIÓN DEL CONTROLADOR

Para crear un controlador utilizamos el siguiente comando:

```
$ php artisan make:controller ArticleController
```

El controlador se crea en `app/Http/Controllers` y se verá como el ejemplo a continuación:

```
use App\Article;
class ArticleController extends Controller
{
    public function index()
    {
        return Article::all();
    }

    public function show($id)
    {
        return Article::find($id);
    }

    public function store(Request $request)
    {
        return Article::create($request->all());
    }

    public function update(Request $request, $id)
    {
        $article = Article::findOrFail($id);
        $article->update($request->all());

        return $article;
    }

    public function delete(Request $request, $id)
    {
        $article = Article::findOrFail($id);
        $article->delete();

        return 204;
    }
}
```

MODIFICAR LAS RUTAS PARA QUE APUNTEN AL CONTROLADOR:

Una vez que tenemos el controlador, podemos modificar las rutas en `routes/api.php` para que sean más sencillas y apunten a nuestro controlador de la siguiente manera:

```
Route::get('articles', 'ArticleController@index');
Route::get('articles/{id}', 'ArticleController@show');
Route::post('articles', 'ArticleController@store');
Route::put('articles/{id}', 'ArticleController@update');
Route::delete('articles/{id}', 'ArticleController@delete');
```

TEST ON POSTMAN

MEJORAR LOS ENDPOINTS MEDIANTE LA VINCULACIÓN IMPLICITA ENTRE EL MODELO Y LA RUTA

De esta manera Laravel inyectará la instancia de `Article` en nuestros métodos y retornará un error 404 si no encuentra el objeto. El archivo `routes/api.php` ahora se verá así:

```
Route::get('articles', 'ArticleController@index');
Route::get('articles/{article}', 'ArticleController@show');
Route::post('articles', 'ArticleController@store');
Route::put('articles/{article}', 'ArticleController@update');
Route::delete('articles/{article}', 'ArticleController@delete');
```

Y el controlador deberá verse así:

```
class ArticleController extends Controller
{
    public function index()
    {
        return Article::all();
    }

    public function show(Article $article)
    {
        return $article;
    }

    public function store(Request $request)
    {
        $article = Article::create($request->all());

        return response()->json($article, 201);
    }

    public function update(Request $request, Article $article)
    {
        $article->update($request->all());

        return response()->json($article, 200);
    }

    public function delete(Article $article)
    {
        $article->delete();

        return response()->json(null, 204);
    }
}
```

TEST ON POSTMAN

PUSH TO GITHUB

CROSS-ORIGIN RESOURCE SHARING (CORS)

En los clientes (navegadores), se prohíbe estrictamente cargar recursos de servidores externos. Sin embargo, si se configura ambos servidores para que trabajen juntos, se puede realizar esta carga. El cross-origin resource sharing (CORS) regula esta colaboración. Para mayor detalle sobre CORS se puede consultar en: https://developer.mozilla.org/es/docs/Web/HTTP/Access_control_CORS o https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/cross-origin-resource-sharing/

Es importante que nuestra API esté en capacidad de controlar que clientes van a poder utilizar los recursos. Para esto se debe crear un middleware en Laravel que permita configurar el CORS. Para esto usamos el siguiente comando:

```
$ php artisan make:middleware Cors
```

El middleware se creará en `app/Http/Middleware/Cors.php` y deberá tener el siguiente código:

```
<?php

namespace App\Http\Middleware;
use Closure;

class Cors
{
  public function handle($request, Closure $next)
  {
    $request->header('Access-Control-Allow-Origin', '*');
    $request->header('Access-Control-Allow-Methods', 'GET, POST, PUT, PATCH,
DELETE, OPTIONS');
    $request->header('Access-Control-Allow-Headers', 'Content-Type,
Authorization');
    return $next($request);
  }
}
```

El middleware establece cuales son los orígenes de las peticiones que podrán utilizar los recursos mediante la cabecera `Access-Control-Allow-Origin`, el `*` representa a todos los orígenes. La cabecera `Access-Control-Allow-Methods` determina que métodos HTTP son aceptados para estas peticiones.

El middleware debe ser registrado en `app/Http/kernel.php` de la siguiente manera:

```
protected $routeMiddleware = [
  'auth' => \App\Http\Middleware\Authenticate::class,
  'auth.basic' =>
\Illuminate\Auth\Middleware\AuthenticateWithBasicAuth::class,
  'guest' => \App\Http\Middleware\RedirectIfAuthenticated::class,
  'cors' => \App\Http\Middleware\Cors::class,
 ];
```

TEST ON POSTMAN

PUSH TO GITHUB

AUTENTICACIÓN: LARAVEL Y JWT

Referencia: https://jwt.io/introduction/

¿QUÉ ES JWT?

JSON Web Token (JWT) es un estándar abierto (RFC-7519) basado en JSON para crear un token que sirva para enviar datos entre aplicaciones o servicios como un objeto JSON y garantizar que sean válidos y seguros. Esta información se puede verificar y confiar porque está firmada digitalmente. Los JWT pueden firmarse usando un secreto (con el algoritmo HMAC) o un par de claves pública / privada usando RSA o ECDSA .

Aunque los JWT se pueden cifrar para proporcionar también secreto entre las partes, nos centraremos en los tokens firmados. Los tokens firmados pueden verificar la integridad de los reclamos que contiene, mientras que los tokens encriptados ocultan esos reclamos de otras partes. Cuando los tokens se firman utilizando pares de claves públicas / privadas, la firma también certifica que solo la parte que posee la clave privada es la que la firmó.

Estos son algunos escenarios en los que los JSON web tokens son útiles:

1. **Autorización**: este es el escenario más común para usar JWT. Una vez que el usuario haya iniciado sesión, cada solicitud posterior incluirá el JWT, lo que le permitirá acceder a rutas, servicios y recursos que están permitidos con ese token. El inicio de sesión único (o Single Sign On) es una característica que utiliza ampliamente JWT hoy en día, debido a su pequeña sobrecarga y su capacidad de usarse fácilmente en diferentes dominios.
2. **Intercambio de información**: los JSON web tokens son una buena forma de transmitir información de forma segura entre las partes. Debido a que los JWT pueden firmarse, por ejemplo, utilizando pares de claves públicas / privadas, puede estar seguro de que los remitentes son quienes dicen ser. Además, como la firma se calcula utilizando el encabezado y la carga útil, también puede verificar que el contenido no haya sido alterado.

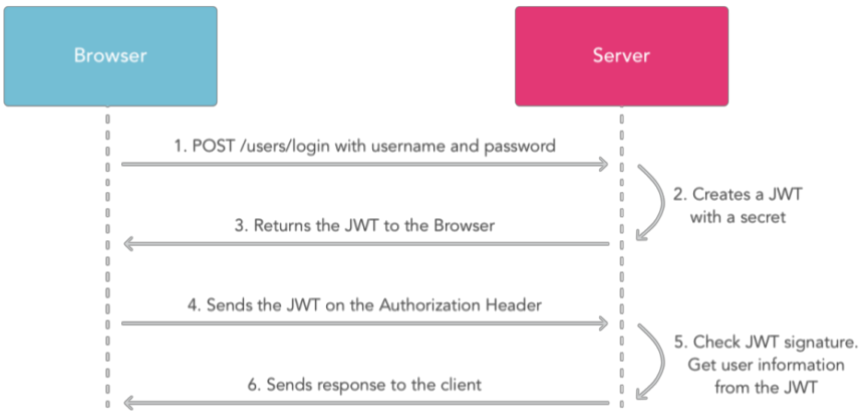

Ilustración 5 Funcionamiento de JWT

El token es enviado en la cabecera `Authorization: Bearer <token>`.

Es muy importante tener en cuenta que, con los tokens firmados, toda la información contenida en el token está expuesta a usuarios u otras partes, a pesar de que no pueden cambiarlo. Por lo tanto, no se debe poner información secreta dentro del token.

ESTRUCTURA DE UN JWT

Los JWT tienen una estructura definida y estándar basada en tres partes, se puede acceder a https://jwt.io/#debugger-io para observar mejor su estructura:

`header.payload.signature`

Las primeras dos partes (`header` y `payload`) son strings en base64 creados a partir dos JSON. La tercera parte (`signature`) toma las otras dos partes y las encripta usando un algoritmo (normalmente SHA-256). Ejemplo:

`eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjEiLCJ1c2VybmFtZSI6ImNoYWxvc2FsdmFkb3IifQ.mONicBrr_eXXFhnsIsGUXoxkNSIn4WR76CjkE7dE4IE`

HEADER

El `header` de un JWT tiene la siguiente forma:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

La propiedad `alg` indica el algoritmo de firma que se utiliza, como HMAC SHA256 o RSA. `typ` define el tipo de token, en nuestro caso `JWT`.

Este JSON se codifica en Base64Url para formar la primera parte del JWT.

PAYLOAD

El `payload` de un JWT es un JSON que puede tener cualquier propiedad, aunque hay una serie de nombres de propiedades definidos en el estándar. Por ejemplo, el JWT mostrado en el ejemplo anterior contiene este payload:

```
{
  "id": "1",
  "username": "chalosalvador"
}
```

De igual manera, el `payload` se codifica en Base64Url para formar la segunda parte del JSON Web Token.

PROPIEDADES ESTÁNDAR

- Creador (`iss`)
- Razón (`sub`)
- Audiencia (`aud`)
- Tiempo de expiración (`exp`)
- No antes (`nbf`)
- Creado (`iat`)
- ID (`jti`)

SIGNATURE

Por último, la firma del JWT se genera usando los anteriores dos campos en base64 y una key secreta (que solo se sepa en los servidores que creen o usen el JWT) para usar con el algoritmo de encriptación. La forma de hacerlo entonces sería la siguiente:

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

La firma se utiliza para verificar que el mensaje no se modificó en el camino y, en el caso de los tokens firmados con una clave privada, también puede verificar que el remitente del JWT es quien dice ser.

INSTALAR JWT

Para empezar a trabajar con JWT en nuestra API, debemos instalarlo con el siguiente comando:

```
$ composer require tymon/jwt-auth:dev-develop --prefer-source
```

Una vez instalado se modifica el archivo `config/app.php` para añadir los providers y los facades de la siguiente manera:

En el arreglo de `providers` añadimos:

```
    [...]
    Tymon\JWTAuth\Providers\LaravelServiceProvider::class,
    [...]
```

En el arreglo de `aliases` se añade lo siguiente:

```
    [...]
    'JWTAuth' => Tymon\JWTAuth\Facades\JWTAuth::class,
    'JWTFactory' => Tymon\JWTAuth\Facades\JWTFactory::class,
    [...]
```

Ahora se debe publicar el archivo de configuración de JWT con el siguiente comando:

```
$ php artisan vendor:publish --
provider="Tymon\JWTAuth\Providers\LaravelServiceProvider"
```

Luego, ejecutamos el comando para generar el jwt-auth secret:

```
$ php artisan jwt:secret
```

Ahora, editamos el modelo `User` para que implemente `JWTSubject`. Debemos definir dos métodos para retornar el `JWTIdentifier` y `JWTCustomClaims`. Los custom claims son utilizados al generar el JWT. El archivo `app/User.php` debe verse así:

```
<?php

  namespace App;

  use Illuminate\Notifications\Notifiable;
  use Illuminate\Foundation\Auth\User as Authenticatable;
  use Tymon\JWTAuth\Contracts\JWTSubject;

  class User extends Authenticatable implements JWTSubject
  {
    use Notifiable;
```

```
    /**
     * Los atributos que pueden ser asignados masivamente.
     *
     * @var array
     */
    protected $fillable = [
        'name', 'email', 'password',
    ];

    /**
     * Los atributos que deben ser ocultos.
     *
     * @var array
     */
    protected $hidden = [
        'password', 'remember_token',
    ];

    public function getJWTIdentifier()
    {
        return $this->getKey();
    }

    public function getJWTCustomClaims()
    {
        return [];
    }
}
```

CREANDO LOS CONTROLADORES PARA USER

Una vez que hemos configurado el modelo `User`, podemos pasar a generar los controladores:

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

Esto generará el archivo `app/Http/UserControllers` que lo debemos editar de la siguiente manera:

```
<?php

namespace App\Http\Controllers;

use App\User;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Hash;
use Illuminate\Support\Facades\Validator;
use JWTAuth;
use Tymon\JWTAuth\Exceptions\JWTException;

class UserController extends Controller
{
    public function authenticate(Request $request)
    {
        $credentials = $request->only('email', 'password');

        try {
            if (! $token = JWTAuth::attempt($credentials)) {
                return response()->json(['error' => 'invalid_credentials'], 400);
            }
        } catch (JWTException $e) {
            return response()->json(['error' => 'could_not_create_token'], 500);
        }

        return response()->json(compact('token'));
    }
```

```
    public function register(Request $request)
    {
      $validator = Validator::make($request->all(), [
        'name' => 'required|string|max:255',
        'email' => 'required|string|email|max:255|unique:users',
        'password' => 'required|string|min:6|confirmed',
      ]);

      if($validator->fails()){
        return response()->json($validator->errors()->toJson(), 400);
      }

      $user = User::create([
         'name' => $request->get('name'),
         'email' => $request->get('email'),
         'password' => Hash::make($request->get('password')),
      ]);

      $token = JWTAuth::fromUser($user);

      return response()->json(compact('user','token'),201);
    }

    public function getAuthenticatedUser()
    {
      try {
        if (! $user = JWTAuth::parseToken()->authenticate()) {
          return response()->json(['user_not_found'], 404);
        }

      } catch (Tymon\JWTAuth\Exceptions\TokenExpiredException $e) {
        return response()->json(['token_expired'], $e->getStatusCode());

      } catch (Tymon\JWTAuth\Exceptions\TokenInvalidException $e) {
        return response()->json(['token_invalid'], $e->getStatusCode());
      } catch (Tymon\JWTAuth\Exceptions\JWTException $e) {
        return response()->json(['token_absent'], $e->getStatusCode());
      }

      return response()->json(compact('user'));
    }
  }
```

El método `authenticate` intenta iniciar la sesión del usuario y genera y token de autorización si el usuario se encuentra en la base de datos. El método lanza un error si no se encuentra el usuario en la base de datos o si se produce alguna excepción.

El método `register` intenta valida los datos de un nuevo usuario y lo crea en la base de datos y devuelve un token de acceso, de esta manera el usuario creado puede iniciar acceder de manera inmediata sin necesidad de iniciar sesión.

El método `getAuthenticatedUser` retorna el objeto del usuario basado en el token de autorización recibido.

CREANDO LAS RUTAS

Antes de crear las rutas de la API, es necesario crear el `JWTMiddleware` que servirá para proteger las rutas. Para esto, se debe correr el siguiente comando:

```
$ php artisan make:middleware JwtMiddleware
```

Esto creará un nuevo archivo en el directorio `app/Http/Middleware`. Se debe ingresar el siguiente contenido en este nuevo archivo:

```
<?php

namespace App\Http\Middleware;

use Closure;
use JWTAuth;
use Exception;
use Tymon\JWTAuth\Exceptions\JWTException;
use Tymon\JWTAuth\Exceptions\TokenExpiredException;
use Tymon\JWTAuth\Exceptions\TokenInvalidException;
use Tymon\JWTAuth\Http\Middleware\BaseMiddleware;

class JwtMiddleware extends BaseMiddleware
{

    /**
     * Handle an incoming request.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        try {
            $user = JWTAuth::parseToken()->authenticate();
        } catch (TokenExpiredException $e) {
            return response()->json(['error' => 'token_expired'], 401);
        } catch (TokenInvalidException $e) {
            return response()->json(['error' => 'token_invalid'], 401);
        } catch (JWTException $e) {
            return response()->json(['error' => 'token_absent'], 401);
        } catch (Exception $e) {
            return response()->json(['error' => $e->getMessage()], 500);
        }
        return $next($request);
    }
}
```

Este middleware extiende a `Tymon\JWTAuth\Http\Middleware\BaseMiddleware`, esto permite que se pueda capturar los errores del token y devolver códigos de error HTTP apropiados al cliente.

Lo siguiente que se debe hacer es registrar el middleware en `app/Http/Kernel.php`:

```
    [...]
    protected $routeMiddleware = [
        [...]
        'jwt.verify' => \App\Http\Middleware\JwtMiddleware::class,
    ];
    [...]
```

Con esto procedemos a crear las rutas en `routes/api.php`:

```
    [...]
    Route::post('register', 'UserController@register');
    Route::post('login', 'UserController@authenticate');
    Route::get('articles', 'ArticleController@index');

    Route::group(['middleware' => ['jwt.verify']], function() {
      Route::get('user', 'UserController@getAuthenticatedUser');
      Route::get('articles/{article}', 'ArticleController@show');
```

```
    Route::post('articles', 'ArticleController@store');
    Route::put('articles/{article}', 'ArticleController@update');
    Route::delete('articles/{article}', 'ArticleController@delete');
});
```

Estas rutas servirán para comprobar el funcionamiento mediante Postman. Las rutas que no se desean proteger, deben quedar fuera del JWT middleware.

PRUEBAS DE JWT EN POSTMAN

A continuación, se presentan varios casos de pruebas con Postman para las rutas de autenticación de nuestra API.

CREAR UNA CUENTA DE USUARIOS PARA PRUEBAS

Para comprobar que se puede registrar nuevos usuarios ingresamos los siguientes datos en Postman:

Endpoint : `127.0.0.1:8000/api/register`
Method: `POST`
Payload:

```
{
  "name": "Usuario Prueba",
  "email": "usuario@prueba.com",
  "password": "123123",
  "password_confirmation": "123123"
}
```

LOGIN DE USUARIO

Ahora que contamos con un usuario registrado, se puede comprobar el inicio de sesión ingresando los siguientes datos en Postman:

Endpoint : `127.0.0.1:8000/api/login`
Method: `POST`
Payload:

```
{
  "email": "usuario@prueba.com",
  "password": "123123"
}
```

ACCEDER UN ENDPOINT NO PROTEGIDO

Para acceder a un endpoint fuera del JWT middleware (no protegido), se ingresa lo siguiente:

Endpoint : `127.0.0.1:8000/api/articles`
Method: `GET`

ACCEDER A UN ENDPOINT PROTEGIDO

Para comprobar que el middleware funciona correctamente se debe ingresar los siguientes datos:

Endpoint : `127.0.0.1:8000/api/articles/1`
Method: `GET`
Payload:

```
Authorization: Bearer INSERTAR_EL_TOKEN_GENERADO
```

OBTENER LOS DATOS DEL USUARIO AUTENTICADO

Endpoint : `127.0.0.1:8000/api/user`
Method: `GET`
Payload:

```
Authorization: Bearer INSERTAR_EL_TOKEN_GENERADO
```

USAR UN TOKEN NO VÁLIDO PARA ACCEDER A LOS DATOS

Endpoint : `127.0.0.1:8000/api/user`
Method: `GET`
Payload:

```
Authorization: Bearer esteesuntokennovalido
```

ACCEDIENDO A UNA RUTA PROTEGIDA SIN UN TOKEN

Endpoint : `127.0.0.1:8000/api/articles/1`
Method: `GET`

PUSH TO GITHUB

ELOQUENT: RELACIONES

Referencia: https://laravel.com/docs/6.x/eloquent-relationships

El modelo de la base de datos del proyecto que se está desarrollando es el siguiente:

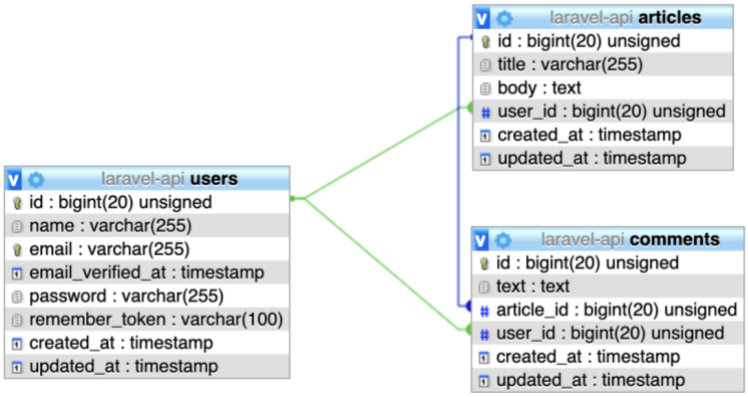

Ilustración 6 Relaciones entre entidades

Sin embargo, hasta ahora no se han implementado las relaciones entre las entidades, las relaciones que se pueden ver son:

- Un usuario puede escribir varios artículos (one to many), la inversa es: Un artículo pertenece a un usuario.
- Un usuario puede escribir varios comentarios (one to many), la inversa es: Un comentario pertenece a un usuario.
- Un artículo puede recibir varios comentarios (one to many), la inversa es: Un comentario pertenece a un artículo.

Para establecer estas relaciones, primero debemos completar la implementación de la base de datos y modificar las tablas para que incluyan las claves foráneas de las relaciones.

MODELANDO LA RELACIÓN ENTRE USER Y ARTICLE

Para establecer la relación entre `User` y `Article` debemos hacer lo siguiente:

1. Añadir las columnas para la clave foránea en `Article`, para esto debemos añadir una nueva migración que modifica la tabla `articles`, con el siguiente comando:

```
php artisan make:migration add_user_id_column_article
```

2. En esta migración hacemos referencia a la tabla `articles` y le añadimos la columna `user_id` e indicamos que esta es una clave foránea de `users`. La migración en el archivo `database/migrations/fecha_hora_add_user_id_column_article.php` debe verse así:

```
class AddUserIdColumnArticle extends Migration
{
  public function up()
  {
    Schema::table('articles', function (Blueprint $table) {
     $table->unsignedBigInteger('user_id');
     $table->foreign('user_id')->references('id')->on('users')-
>onDelete('restrict');
```

```
    });
  }

  public function down()
  {
    Schema::table('articles', function (Blueprint $table) {
      $table->dropForeign(['user_id']);
    });
  }
}
```

up():

Las siguientes líneas añaden la columna `user_id` a la tabla `articles` y establecen la clave foránea con la columna `id` de `users`. El método `onDelete('restrict')` impone la restricción al intentar eliminar el registro padre de la relación. Si se usa el valor `'cascade'`, al eliminar el registro padre, se eliminarán todos los hijos.

```
$table->unsignedBigInteger('user_id');
$table->foreign('user_id')->references('id')->on('users')-
>onDelete('restrict');
```

down():

La siguiente línea elimina la columna `user_id` a la tabla `articles` y la clave foránea:

```
$table->dropForeign(['user_id']);
```

3. Ejecutar la migración.

```
php artisan migrate
```

Después de completar estos pasos, la tabla `articles` en nuestra base de datos debe incluir el campo `user_id` y se debe reflejar la relación entre las entidades.

CREANDO EL MODELO PARA COMMENT

Lo siguiente que debemos hacer es crear la tabla `comments` estableciendo las claves foráneas correspondientes.

1. Ejecutar el comando para crear el modelo `Comment` junto con la migración:

```
php artisan make:model Comment -m
```

2. Este comando creará el modelo para `Comment` y la migración correspondiente, empezaremos completando la migración en el archivo `database/migrations/fecha_hora_create_comments_table.php` de la siguiente manera:

```
class CreateCommentsTable extends Migration
{
  public function up()
  {
    Schema::create('comments', function (Blueprint $table) {
      $table->bigIncrements('id');
      $table->text('text');
      $table->unsignedBigInteger('article_id');
      $table->foreign('article_id')->references('id')->on('articles')-
>onDelete('restrict');
```

```
            $table->unsignedBigInteger('user_id');
            $table->foreign('user_id')->references('id')->on('users')-
>onDelete('restrict');
            $table->timestamps();
        });
    }

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

El método `up()` está creando la tabla y está estableciendo los campos `user_id` y `article_id` como claves foráneas.

3. Por ahora en el modelo `Comment` solo incluimos la siguiente línea:

```
<?php

namespace App;
use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    protected $fillable = ['text'];
}
```

4. Ejecutar la migración, después de ejecutar el siguiente comando, se creará la tabla `comments` en nuestra base de datos con las relaciones establecidas.

```
php artisan migrate
```

Con esto tenemos nuestra base de datos lista, ahora debemos configurar los modelos para poder utilizar la *"magia"* de Eloquent. Para esto vamos a revisar cada uno de los tipos de relaciones y los métodos que Eloquent nos proporciona:

ONE TO MANY

En el sistema se tienen varias relaciones uno a muchos, para hacer las consultas de manera súper sencilla, Eloquent nos proporciona el método `hasMany()` que debemos añadir en nuestros modelos como lo vemos a continuación:

1. **Un usuario puede escribir varios artículos**: Para reflejar esta relación se debe añadir el método `articles()` al final del modelo `User`:

```
[...]

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

[...]
```

Eloquent determinará automáticamente la columna adecuada para la clave foránea en el modelo `Article`. Por convención, Eloquent tomará la forma "snake_case" del nombre del modelo padre con el sufijo `_id`. Entonces para este caso la clave foránea en el modelo `Article` será `user_id`.

Si se desea especificar manualmente las columnas se las puede pasar como parámetros así: `hasMany('App/Model', 'foreign_key', 'local_key')`, ejemplo:

```
$this->hasMany('App\Article','user_id');

$this->hasMany('App\Article','user_id','id');
```

Una vez que se establece la relación, se puede acceder a la colección de artículos de un usuario mediante la propiedad `articles` del modelo `User`. A esto se le conoce como las **propiedades dinámicas** que son proporcionadas por Eloquent, esto permite acceder a los métodos de las relaciones como si estuvieran definidos como propiedades en el modelo.

```
$articles = App\User::find(1)->articles;

foreach ($articles as $article) {
    //
}
```

Debido a que las relaciones pueden servir también como constructores de consultas (query builders), se pueden añadir más restricciones para determinar que artículos deben ser devueltos al llamar al método `articles` y añadiendo condiciones en cadena a la consulta.

```
$article = App\User::find(1)->articles()->where('title', 'foo')->first();
```

2. **Un usuario puede escribir varios comentarios**: Esta relación la reflejamos de manera similar añadiendo el método `comments` al final del modelo `Users`:

```
public function comments()
{
    return $this->hasMany('App\Comment');

}
```

Las consultas y todo lo explicado en el punto anterior para `articles` se aplican exactamente de la misma manera para `comments`.

3. **Un artículo puede recibir varios comentarios**: Para reflejar esta relación añadimos lo siguiente en el modelo `Article`:

```
public function comments()
{
    return $this->hasMany('App\Comment');
}
```

La única diferencia en este caso es que la clave foránea que se establece es `article_id` que está presente en la tabla `comments`, ya que el modelo en el que lo estamos implementando es `Article`.

ONE TO MANY (INVERSA)

Es conveniente poder acceder a la relación de manera inversa, para esto Eloquent nos brinda el método `belongsTo()` que debe añadirse en los modelos hijos. Como lo vimos al inicio de la sección, las relaciones inversas correspondientes serían las siguientes:

1. **Un artículo pertenece a un usuario:** Esta relación inversa permite que un artículo pueda acceder al objeto del usuario propietario. En este caso sería el modelo hijo sería `Article`, entonces se debe añadir el siguiente método en este modelo:

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

De igual manera, Eloquent determina automáticamente la columna y la clave foránea, pero también se lo puede hacer de manera explicita:

```
$this->belongsTo('App\User','user_id');

$this->belongsTo('App\User','user_id','id');
```

Una vez que se establece esta relación, se puede acceder al objeto del modelo `User` para un `Article` accediendo a la "propiedad dinámica" `user`:

```
$article = App\Article::find(1);

echo $article->user->email;
```

En el caso de que se haya establecido el nombre de la clave foránea diferente a la que se establece por defecto (`modelname_id`), se puede pasar el nombre de la clave foránea al método `belongsTo`, de esta manera Eloquent podrá saber como se ha establecido esta relación y devolverá el objeto correspondiente.

```
$this->belongsTo('App\User', 'foreign_key');
```

De igual manera, si el modelo padre no utiliza `id` como su clave primaria o se desea establecer la relación con una columna diferente, se lo puede indicar como un tercer parámetro al método `belongsTo`.

```
$this->belongsTo('App\User', 'foreign_key', 'other_key');
```

2. **Un comentario pertenece a un usuario:** Si deseamos acceder al objeto del usuario que escribió un comentario, entonces añadimos lo siguiente al modelo `Comment`:

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

3. **Un comentario pertenece a un artículo:** Por último, si queremos acceder al artículo al cual pertenece un comentario, añadimos lo siguiente en `Comment`:

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

OBTENER EL ID DEL USUARIO CON SESIÓN ACTIVA

Cuando queremos crear un nuevo artículo o comentario, se debe obtener de manera automática el id del usuario con la sesión activa o el usuario al que pertenece el token recibido, esto con el objetivo de establecer que usuario está creando este registro y establecer el valor de la clave foránea. Veremos dos maneras de hacer esto.

1. AUTH FACADE

Una manera de conseguir esto es utilizando el método `boot()` en el modelo `Article` de la siguiente manera:

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

    static::creating(function ($article) {
        $article->user_id = Auth::id();
    });
}
```

Podemos insertar el mismo método en el modelo `Comment` para obtener de manera automática el `id` de usuario que realiza la petición.

El método `boot` se ejecuta permite personalizar el comportamiento de los eventos en un modelo, en este caso estamos diciendo que al momento de crear un artículo le asigne a la propiedad `user_id` el `id` del usuario que tiene la sesión actual. Para obtener el id del usuario actual estamos utilizando el **Auth Facade** que proporciona Laravel. Para conocer más sobre los facades se puede revisar la documentación aquí: https://laravel.com/docs/6.x/facades y para conocer más sobre el Auth facade aquí: https://laravel.com/docs/6.x/authentication#retrieving-the-authenticated-user.

2. VIA EL CONTROLADOR

Podemos lograr lo mismo si modificamos ligeramente las acciones store de los controladores de `Article` y `Comment`, así:

```
public function store(Request $request)
{
    $article = new Article($request->all());
    $article->user_id = Auth::id();
    $article->save();
    return response()->json(new ArticleResource($article), 201);
}
```

Sin embargo, al implementar esto en el controlador no tendría efecto al momento de crear el seeder por lo que en los seeders tendríamos que implementarlo de manera manual añadiendo el atributo `user_id`:

```
Article::create([
    'title' => $faker->sentence,
    'body' => $faker->paragraph,
    'user_id' => Auth::id()
]);
```

CREAR EL SEEDER TOMANDO EN CUENTA LAS RELACIONES Y LA SESIÓN ACTIVA

Para poder empezar a probar nuestro sistema de una manera muy rápida, debemos modificar los seeders que tenemos para que se reflejen las relaciones que tenemos.

El seeder para `User`, necesita ninguna modificación.

El método `run` para el seeder de `Article`, ahora deberá simular la sesión de los usuarios que hay en la base de datos (generados por el seeder de `Users`), entonces debe quedar de la siguiente manera:

```
public function run()
{
```

```
    // Vaciar la tabla articles.
    Article::truncate();
    $faker = \Faker\Factory::create();

    // Obtenemos la lista de todos los usuarios creados e
    // iteramos sobre cada uno y simulamos un inicio de
    // sesión con cada uno para crear artículos en su nombre
    $users = App\User::all();
    foreach ($users as $user) {
        // iniciamos sesión con este usuario
        JWTAuth::attempt(['email' => $user->email, 'password' => '123123']);

        // Y ahora con este usuario creamos algunos articulos
        $num_articles = 5;
        for ($j = 0; $j < $num_articles; $j++) {
            Article::create([
                'title' => $faker->sentence,
                'body' => $faker->paragraph,
            ]);
        }

    }
}
```

Ahora debemos crear el seeder para `Comment`, ejecutamos el comando:

```
php artisan make:seeder CommentsTableSeeder
```

El método `run` para este seeder se verá así:

```
public function run()
{
    // Vaciamos la tabla comments
    Comment::truncate();
    $faker = \Faker\Factory::create();

    // Obtenemos todos los artículos de la bdd
    $articles = App\Article::all();

    // Obtenemos todos los usuarios
    $users = App\User::all();
    foreach ($users as $user) {
        // iniciamos sesión con cada uno
        JWTAuth::attempt(['email' => $user->email, 'password' => '123123']);

        // Creamos un comentario para cada artículo con este usuario
        foreach ($articles as $article) {
            Comment::create([
                'text' => $faker->paragraph,
                'article_id' => $article->id,
            ]);
        }
    }
}
```

Para nuestro caso que tenemos modelos sencillos, es suficiente utilizar los seeders de esta manera, en caso de tener mayor cantidad de modelos y más complejos, es recomendable utilizar los factories: https://laravel.com/docs/6.x/database-testing#generating-factories.

UTILIZACIÓN DE LARAVEL TINKER

Para probar que nuestro sistema está funcionando correctamente con los datos ficticios que hemos generado, podemos utilizar **Tinker**, el cual es un REPL incorporado en Laravel que nos permite interactuar con nuestro sistema desde la línea de comandos. REPL significa Read-Eval-Print-Loop (Bucle Lectura-Evaluación-Impresión).

Para acceder a Tinker debemos asegurarnos que tenemos instalada la dependencia `laravel/tinker` en nuestro `composer.json` y ejecutar el siguiente comando:

```
php artisan tinker
```

Una vez dentro de Tinker podremos declarar variables y acceder a nuestros modelos mediante y ejecutar los métodos de Eloquent, por ejemplo:

- Si escribimos lo siguiente en la línea de comandos, nos presentará un arreglo con todos los artículos en la base de datos.

  ```
  $articles = App\Article::all();
  ```

- Luego podemos acceder a la información del usuario que creó el primer artículo del arreglo, escribiendo los siguiente en la línea de comandos:

  ```
  $articles[0]->user
  ```

  Esto lo hace utilizando el método `user()` que incluimos en el modelo `Article`.

- También podemos acceder a los comentarios que ha recibido el artículo de la siguiente manera:

  ```
  $articles[0]->comments
  ```

  Para esto utiliza el método `comments()` que se incluyó en el método `Article`

- De igual manera podemos acceder a todos los artículos escritos por el usuario que tiene el `id = 1`, así:

  ```
  $articles = App\User::find(1)->articles
  ```

  En este caso utiliza el método `articles()` incluido en el modelo `User`.

MANY TO MANY

Las relaciones muchos a muchos son ligeramente más complejas de lo que hemos visto hasta ahora, tomando en cuenta lo simple que Laravel hace todas estas tareas.

Un ejemplo de esta relación en el contexto del sistema que estamos desarrollando, sería un usuario que puede suscribirse a varías de las categorías de artículos (para recibir notificaciones vía correo más adelante). Así mismo, una categoría podría recibir suscripciones de varios usuarios, en este caso tenemos una relación de muchos a muchos.

ESTRUCTURA DE LAS TABLAS

Nuestro modelo de la base se modificaría, aumentando una tabla `categories` y las relaciones con `articles` y con `users`. Además, necesitamos una tabla intermedia llamada `category_user` que rompe la relación de muchos a muchos entre `users` y `categories`. El nombre de esta tabla intermedia se deriva del orden alfabético de los modelos relacionados y debe contener las columnas `user_id` y `category_id`. Entonces quedaría así:

```
users
    id - integer
    name - string
    // el resto de columnas de users
```

```
categories
    id - integer
    name - string

category_user
    user_id - integer
    category_id - integer
    created_at - timestamp

articles
    id
    category_id
    // el resto de columnas de articles
```

En este caso la columna `created_at` en la tabla `category_user` nos dirá la fecha en la que el usuario se suscribió a la categoría.

Entonces, nuestro modelo de base datos quedaría así:

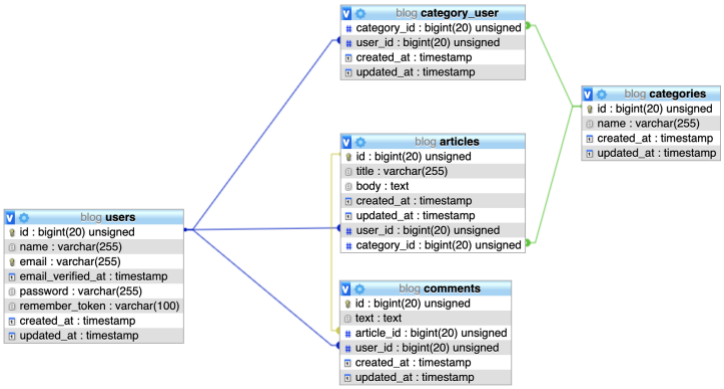

Ilustración 7 Modelo de bdd con categories

Las nuevas relaciones serían:

- Una categoría puede tener varios artículos (one to many), la inversa es: Un artículo pertenece a una categoría.
- Un usuario puede suscribirse a varias categorías y una categoría puede tener varios usuarios. Por lo tanto, esta es una relación de **muchos a muchos** que se rompe con la tabla intermedia `category_user`. El nombre `category_user` se deriva de la convención de utilizar los nombres de las tablas involucradas en orden alfabético.
  Otra manera de ver esta relación es viendo a la tabla `category_user` como subscripciones de usuarios. Así, la relación puede ser leída como: Un usuario puede tener varias subscripciones y una subscripción pertenece a un usuario (one to many). De igual manera, una categoría pertenece a varias subscripciones y una subscripción pertenece a una categoría (one to many).

ESTRUCTURA DEL MODELO

Entonces, para incluir la tabla `categories` y las nuevas relaciones, debemos crear el nuevo modelo con su migración:

```
php artisan make:model Category -m
```

Esta migración tendrá todo lo relacionado con la tabla `categories`, es decir, crear la tabla `categories`, crear la tabla `category_user` con las relaciones para `categories` y `users` y, además, la migración modifica la tabla `articles` para añadir la clave foránea `category_id`.

```
class CreateCategoriesTable extends Migration
{
    public function up()
    {
        Schema::create('categories', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('name');
            $table->timestamps();
        });

        Schema::create('category_user', function (Blueprint $table) {
            $table->unsignedBigInteger('category_id');
            $table->foreign('category_id')->references('id')-
>on('categories')->onDelete('restrict');
            $table->unsignedBigInteger('user_id');
            $table->foreign('user_id')->references('id')->on('users')-
>onDelete('restrict');
            $table->timestamps();
        });

        Schema::table('articles', function (Blueprint $table) {
            $table->unsignedBigInteger('category_id')->nullable();
            $table->foreign('category_id')->references('id')-
>on('categories')->onDelete('restrict');
        });
    }

    public function down()
    {
        Schema::disableForeignKeyConstraints();
        Schema::dropIfExists('category_user');
        Schema::dropIfExists('categories');
        Schema::table('articles', function (Blueprint $table) {
            $table->dropForeign('category_id');
        });
        Schema::enableForeignKeyConstraints();
    }
}
```

Las relaciones muchos a muchos en Eloquent se definen mediante un método que retorna el resultado del método `belongsToMany`. Por ejemplo, el método `categories` en el modelo `User` sería así:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The categories that belong to the user.
     */
    public function categories()
    {
        return $this->belongsToMany('App\Category');
    }
}
```

De igual manera que en las relaciones one to many, podemos acceder a las categorías de un usuario mediante la propiedad dinámica `categories`.

```
$user = App\User::find(1);
foreach ($user->categories as $category) {
    //
}
```

También se puede encadenar las condiciones de las consultas directo a la llamada de la relación:

```
$categories = App\User::find(1)->categories()->orderBy('name')->get();
```

Eloquent determina el nombre de la tabla intermedia uniendo en orden alfabético los nombres de los modelos relacionados. Si se desea utilizar otro nombre se lo puede pasar como parámetro al método `belongsToMany`. **Es importante notar que dentro de Laravel no hace falta crear un modelo para** `category_user`.

```
return $this->belongsToMany('App\Category', 'category_user');
```

También se puede modificar los nombres de las columnas utilizadas como claves foráneas en la tabla intermedia. El tercer parámetro para el método `belongsToMany` es el nombre de la clave foránea en el cual se está definiendo la relación, mientras que el cuarto parámetro es el nombre de la clave foránea del modelo con el que se lo está relacionando.

```
return $this->belongsToMany('App\Category', 'category_user', 'user_id',
'category_id');
```

MANY TO MANY (INVERSA)

Para definir la relación inversa, se inserta otra llamada a `belongsToMany` en el modelo relacionado, en este caso en `Category`:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Category extends Model
{
    protected $fillable = ['name'];

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

RECUPERANDO COLUMNAS DE LA TABLA INTERMEDIA

Debido a que las relaciones de muchos a muchos necesitan una tabla intermedia, Eloquent proporciona varias maneras de interactuar con esta tabla. Por ejemplo, si asumimos que un usuario se ha suscrito a varias categorías, y que estamos guardando la fecha en la que el usuario se suscribió, podemos acceder a la columna `created_at` utilizando el atributo `pivot` en los modelos:

```
$user = App\User::find(1);

foreach ($user->categories as $category) {
    echo $category->pivot->created_at;
}
```

Para poder llevar los campos `created_at` y `updated_at` de manera automática en nuestra tabla pivote, podemos utilizar el método `withTimestamps` al definir la relación:

```
return $this->belongsToMany('App\Category')->withTimestamps();
```

Por defecto, solo las llaves primarias de los modelos formarán parte del objeto pivote. Si se desea incluir columnas extra, se las debe especificar al definir la relación:

```
return $this->belongsToMany('App\Category')->withPivot('column1',
'column2');
```

PERSONALIZANDO EL NOMBRE DEL ATRIBUTO PIVOTE

Se puede modificar el nombre del atributo `pivot` para reflejar de mejor manera su propósito. Por ejemplo, en nuestro caso un nombre más adecuado podría ser `subscriptions`:

```
return $this->belongsToMany('App\Category')
                ->as('subscriptions')
                ->withTimestamps();
```

Ahora podemos acceder a nuestra tabla intermedia usando el nombre personalizado:

```
$user = App\User::find(1);

foreach ($user->categories as $category) {
    echo $category->subscriptions->created_at;
}
```

O podemos acceder a la lista de todas las subscripciones de todos los usuarios así:

```
$users = User::with('categories')->get();

foreach ($users->flatMap->categories as $category) {
    echo $category->subscriptions->created_at;
}
```

FILTRANDO LAS RELACIONES MEDIANTE COLUMNAS DE LA TABLA INTERMEDIA

Podemos filtrar los resultados de `belongsToMany` utilizando los métodos `wherePivot`, `wherePivotIn`, y `wherePivotNotIn` al definir la relación:

```
return $this->belongsToMany('App\Category')->wherePivot('approved', 1);

return $this->belongsToMany('App\Category')->wherePivotIn('priority', [1,
2]);

return $this->belongsToMany('App\Category')->wherePivotNotIn('priority', [1,
2]);
```

Esta es una pequeña muestra de lo potente que es Eloquent, podemos ver más “trucos” de Eloquent revisando la siguiente lista: https://laravel-news.com/eloquent-tips-tricks.

MODIFICACIÓN DE LOS SEEDERS

Crear un nuevo seeder para `categories`:

```
php artisan make:seeder CategoriesTableSeeder
```

Este seeder es sencillo, solamente debe crear las categorías con nombres ficticios, en este ejemplo estamos agregando 3 categorías:

```
class CategoriesTableSeeder extends Seeder
{
    public function run()
```

```
    {
        // Vaciamos la tabla categories
        Category::truncate();
        $faker = \Faker\Factory::create();

        for ($i = 0; $i < 3; $i++) {
            Category::create([
                'name' => $faker->word
            ]);
        }

    }
}
```

Ahora debemos modificar el seeder para `articles` para relacionar un artículo con una categoría:

```
for ($j = 0; $j < $num_articles; $j++) {
    Article::create([
        'title' => $faker->sentence,
        'body' => $faker->paragraph,
        'category_id' => $faker->numberBetween(1, 3)
    ]);
}
```

En el seeder de `users` podemos hacer que, al crear un usuario, este se subscriba aleatoriamente a diferentes categorías, para esto agregamos lo siguiente:

```
for ($i = 0; $i < 10; $i++) {
    $user = User::create([
        'name' => $faker->name,
        'email' => $faker->email,
        'password' => $password,
    ]);

    $user->categories()->saveMany(
        $faker->randomElements(
            array(
                Category::find(1),
                Category::find(2),
                Category::find(3)
            ), $faker->numberBetween(1, 3), false
        )
    );
}
```

Por último, modificamos el `DatabaseSeeder.php` agregando el seeder de `Categories`:

```
public function run()
{
    Schema::disableForeignKeyConstraints();
    $this->call(CategoriesTableSeeder::class);
    $this->call(UsersTableSeeder::class);
    $this->call(ArticlesTableSeeder::class);
    $this->call(CommentsTableSeeder::class);
    Schema::enableForeignKeyConstraints();
}
```

Corremos el seeder para confirmar que se insertan correctamente los datos de prueba:

```
php artisan de:seed
```

PUSH TO GITHUB

ELOQUENT: RECURSOS API

Referencia: https://laravel.com/docs/6.x/eloquent-resources

Al construir una API, se necesita tener una capa intermedia entre los modelos y las respuestas JSON que se envían al cliente. Laravel proporciona las clases `Resources` que permiten transformar los modelos y las colecciones en JSON de manera sencilla.

GENERANDO RECURSOS

Para generar un recurso se ejecuta el siguiente comando:

```
php artisan make:resource User
```

Los recursos se crean en `app/Http/Resources`.

Todo recurso define un método `toArray` el cual retorna el arreglo de atributos del modelo que debe ser convertido a JSON al enviar la respuesta al cliente. Se puede acceder a las propiedades del modelo directamente mediante la variable `$this`. Esto es porque la clase del recurso se vincula directamente con el modelo. Por ejemplo, la clase recurso para `User` podría verse así:

```
<?php

namespace App\Http\Resources;
use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource
{
    /**
     * Transformar the resource into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'email' => $this->email,
            'created_at' => $this->created_at,
            'updated_at' => $this->updated_at,
        ];
    }
}
```

Entonces los controladores deben retornar el recurso del modelo en lugar de retornar el modelo directamente. Por ejemplo:

```
use App\Http\Resources\User as UserResource;
use App\User;
Route::get('/user', function () {
    return new UserResource(User::find(1));
});
```

O, en el controlador:

```
public function show(Article $article)
{
    return response()->json(new ArticleResource($article), 200);
}
```

COLECCIONES DE RECURSOS

Además de generar recursos para modelos individuales, se puede generar recursos para colecciones de modelos. Esto nos permite incluir enlaces y otros metadatos relevantes a toda la colección de un recurso.

Si se va a retornar una colección de recursos o una respuesta con paginación, se puede utilizar el método `collection` al crear la instancia del recurso en el controlador, así:

```
use App\Http\Resources\User as UserResource;
use App\User;

Route::get('/user', function () {
    return UserResource::collection(User::all());
});
```

Sin embargo, esto no permite la inclusión de metadatos en la respuesta. Si se desea personalizar la respuesta de la colección de recursos, se debe crear un recurso dedicado para representar la colección. Para hacer esto tenemos dos maneras de hacerlo:

1. Añadiendo el parámetro `--collection`

```
php artisan make:resource Users --collection
```

2. Añadiendo la palabra `Collection` al nombre

```
php artisan make:resource UserCollection
```

Entonces, así podemos incluir metadatos en la respuesta:

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\ResourceCollection;

class UserCollection extends ResourceCollection
{
    /**
     * Transform the resource collection into an array.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return array
     */
    public function toArray($request)
    {
        return [
            'data' => $this->collection,
            'links' => [
                'self' => 'link-value',
            ],
        ];
    }
}
```

Si ya tenemos el recurso para la colección, podemos retornarlo desde nuestro controlador:

```
use App\Http\Resources\UserCollection;
use App\User;

Route::get('/users', function () {
    return new UserCollection(User::all());
});
```

PRESERVAR LAS LLAVES EN LAS COLECCIONES

Por defecto, Laravel reinicia las llaves de la colección para que estén en orden numérico. En caso de que se desee preservar las llaves originales se debe añadir la propiedad `preserveKeys`.

```
<?php

namespace App\Http\Resources;

use Illuminate\Http\Resources\Json\JsonResource;

class User extends JsonResource
{
    /**
     * Indicates if the resource's collection keys should be preserved.
     *
     * @var bool
     */
    public $preserveKeys = true;
}
```

INCLUIR LAS RELACIONES EN LOS RECURSOS

Para incluir los modelos relacionados en los recursos, se puede añadirlos al arreglo, por ejemplo, para el modelo `User`, vamos a incluir la colección de `Article` en la respuesta:

```
/**
 * Transform the resource into an array.
 *
 * @param  \Illuminate\Http\Request  $request
 * @return array
 */
public function toArray($request)
{
    return [
        'id' => $this->id,
        'name' => $this->name,
        'email' => $this->email,
        'articles' => ArticleResource::collection($this->articles),
        'created_at' => $this->created_at,
        'updated_at' => $this->updated_at,
    ];
}
```

PAGINACIÓN

Para ver más claramente cuando son útiles los metadatos que podemos incluir en el resultado de una colección, podemos ver como funciona el método `paginate` de un modelo.

```
public function index(Request $request)
{
  return ArticleResource::collection(Article::paginate());
}
```

Al acceder a la ruta `/api/articles` vemos que ahora el resultado es el siguiente:

```
"data": [
...
],
"links": {
  "first": "http://localhost:8000/api/articles?page=1",
  "last": "http://localhost:8000/api/articles?page=4",
```

```
    "prev": null,
    "next": "http://localhost:8000/api/articles?page=2"
},
"meta": {
    "current_page": 1,
    "from": 1,
    "last_page": 4,
    "path": "http://localhost:8000/api/articles",
    "per_page": 15,
    "to": 15,
    "total": 57
}
```

Como vemos, el resultado está paginado y por defecto nos devuelve los primeros 15 artículos, y en los atributos `links` y `meta` incorpora todos los datos necesarios para navegar en la paginación de los artículos. Si deseamos cambiar la cantidad de elementos por página lo podemos pasar como parámetro:

```
ArticleResource::collection(Article::paginate(5));
```

TEST ON POSTMAN

PUSH TO GITHUB

VALIDACIÓN DE DATOS

Referencia: https://laravel.com/docs/6.x/validation

LÓGICA DE VALIDACIÓN

Laravel nos proporciona el método `validate` mediante el objeto `Illuminate\Http\Request`. Si los datos pasan las reglas de validación, se seguirá ejecutando el código normalmente. Pero si la validación falla se lanzará una excepción, enviando la respuesta de error correspondiente al cliente. Para las llamadas API, la petición debe incluir la cabecera `Accept: application/json`, solo así Laravel podrá identificar que es una llamada al API y devolverá una respuesta en formato JSON.

La validación se la puede hacer dentro del método `store` del controlador, pasando las reglas de validación que se desea para cada columna del modelo, por ejemplo:

```
/**
 * Store a new article.
 *
 * @param  Request  $request
 * @return Response
 */
public function store(Request $request)
{
    $validatedData = $request->validate([
        'title' => 'required|string|unique:articles|max:255',
        'body' => 'required',
    ]);

    $article = Article::create($validatedData);
    return response()->json(new ArticleResource($article), 201);
}
```

También se puede pasar las reglas como un arreglo:

```
$validatedData = $request->validate([
    'title' => ['required', 'unique:articles', 'max:255'],
    'body' => ['required'],
]);
```

En este caso, si la validación no pasa, por ejemplo, por no enviar el campo `body`, la respuesta tendrá un código de error `422` que corresponde al mensaje `"Unprocessable Entity"`, indicando que los datos enviados no pueden ser procesados. El cuerpo del error sería como el siguiente:

```
{
  message: "The given data was invalid.",
  errors: {
    body: [
      0: "The body field is required."
    ]
  }
}
```

CREACIÓN MANUAL DE VALIDADORES

Si se desea tener mayor control sobre la respuesta enviada al cliente, se puede crear el validador manualmente utilizando el método `make` del facade `Validator`. Así:

```
public function store(Request $request)
{
    $validator = Validator::make($request->all(), [
      'title' => 'required|string|unique:articles|max:255',
      'body' => 'required|string'
    ]);

    if ($validator->fails()) {
      return response()->json(['error' => 'data_validation_failed',
"error_list"=>$validator->errors()], 400);
    }

    $article = Article::create($request->all());
    return response()->json(new ArticleResource($article), 201);
 }
```

El primer parámetro del método `make` es el arreglo de datos a validar. El segundo parámetro es el arreglo con las reglas de validación que se deben aplicar a los datos.

En este caso, para saber si la validación falló, se utiliza el método `$validator->fails()` y se devuelve una respuesta apropiada de manera manual. Por ejemplo, se puede cambiar el código de error por `400` en lugar de `422`.

Cuerpo de la respuesta para este caso sería:

```
{
  message: "data_validation_failed",
  errors: {
    body: [
      0: "The body field is required."
    ]
  }
}
```

MENSAJES DE ERROR PERSONALIZADOS

Tanto el método `validate` como el método `make`, pueden recibir un parámetro con un arreglo que contiene los errores personalizados.

```
$messages = [
    'required' => 'El campo :attribute es obligatorio.',
];

$validator = Validator::make($input, $rules, $messages);
```

Notar que se utiliza el placeholder `:attribute` que será reemplazado por el nombre del campo correspondiente. El placeholder se lo puede utilizar en varias reglas, por ejemplo:

```
$messages = [
    'same'    => 'Los campos :attribute y :other deben coincidir.',
    'size'    => 'El campo :attribute debe tener exactamente :size.',
    'between' => 'El valor del campo :attribute :input no está entre :min -
:max.',
    'in'      => 'El campo :attribute debe estar entre las siguientes
opciones: :values',
];
```

PERSONALIZAR UN MENSAJE PARA UN CAMPO ESPECÍFICO

Si se desea tener un mensaje personalizado para un campo y una regla específicos, se lo puede hacer con la notación de punto:

```
$validatedData = $request->validate([
    'title' => 'required|unique:articles|max:255',
    'body' => 'required',
], [
    'body.required' => 'Body no valido'
]);
```

REGLAS DE VALIDACIÓN DISPONIBLES

La lista de reglas de validación disponibles se encuentra en la documentación:

Accepted
Active URL
After (Date)
After Or Equal (Date)
Alpha
Alpha Dash
Alpha Numeric
Array
Bail
Before (Date)
Before Or Equal (Date)
Between
Boolean
Confirmed
Date
Date Equals
Date Format
Different
Digits
Digits Between
Dimensions (Image Files)
Distinct
E-Mail
Ends With
Exclude If
Exclude Unless
Exists (Database)
File
Filled
Greater Than
Greater Than Or Equal
Image (File)
In
In Array
Integer
IP Address
JSON
Less Than
Less Than Or Equal
Max
MIME Types
MIME Type By File Extension
Min
Not In
Not Regex
Nullable
Numeric
Password
Present
Regular Expression
Required
Required If
Required Unless
Required With
Required With All
Required Without
Required Without All
Same
Size
Sometimes
Starts With
String
Timezone
Unique (Database)
URL
UUID

TEST ON POSTMAN

PUSH TO GITHUB

IMÁGENES

SUBIR UNA IMAGEN AL SERVIDOR

Recibir una imagen desde el cliente y subirla al servidor es una tarea sencilla con Laravel. Para empezar, vamos a agregar el campo `image` en el modelo `Article`. Para esto seguimos los pasos a continuación:

1. Crear una migración que modifique la tabla `articles`.

```
php artisan make:migration add_image_column_article
```

Esto generará el archivo de la migración en `database/migrations/fecha_hora_add_image_column_article.php`, la migración debe contener lo siguiente:

```
class AddImageColumnArticle extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {
        Schema::table('articles', function (Blueprint $table) {
            $table->string('image');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {
        Schema::table('articles', function (Blueprint $table) {
            $table->dropColumn('image');
        });
    }
}
```

El método `up()` está agregando la columna `image` de tipo `string` a la tabla `articles`. Esto nos permitirá almacenar el nombre del archivo de la imagen que corresponde al artículo.

2. Agregamos la nueva columna en el arreglo `$fillable` del modelo `Article`:

```
protected $fillable = ['title', 'body', 'category_id', 'image'];
```

3. Editamos la acción `store` del `ArticleController`, debe quedar de la siguiente manera:

```
public function store(Request $request)
{
  $request->validate([
    'title' => 'required|string|unique:articles|max:255',
    'body' => 'required',
    'category_id' => 'required|exists:categories,id',
    'image' => 'required|image|dimensions:min_width=200,min_height=200',
  ]);

  $article = new Article($request->all());
  $path = $request->image->store('articles');
```

```
    $article->image = $path;

    $article->save();

    return response()->json(new ArticleResource($article), 201);
}
```

Cabe notar que ya no se hace uso del método `create`, si no que estamos creando una instancia del modelo `Article` con los datos que vienen en el `$request`, luego estamos tomando el valor del campo `image` que llegó en `$request` y mediante el método `store()` estamos subiendo la imagen al servidor.

Por defecto, el método store almacena los archivos en el directorio `storage/app/`, sin embargo, los archivos almacenados aquí no son públicamente accesibles. En este caso, deseamos que nuestra imagen sea de acceso público por lo que estamos pasando como parámetro la ruta del directorio donde queremos almacenar el archivo (`store('public/articles')`), por lo tanto, el archivo se subirá al directorio `storage/app/public/articles/`.

El nombre del archivo que se sube al servidor será generado de manera aleatoria por el método `store`. Si queremos personalizar el nombre del archivo, debemos utilizar el método `storeAs()`, por ejemplo, si queremos que el nombre del archivo sea una combinación del `id del usuario que envía la petición + título del artículo + extensión del archivo`, debemos hacer lo siguiente:

```
$path = $request->image->storeAs('public/articles', $request->user()->id .
'_' . $article->title . '.' . $request->image->extension());
```

- `$request->user()->id` obtiene el id del usuario que envía la petición desde el token.
- `$article->title` obtiene el título del artículo
- `$request->image->extension()` obtiene la extensión del archivo original.

Se recomienda dejar que se genere un nombre aleatorio ya que de lo contrario tendremos que normalizar y limpiar el nombre del archivo antes de almacenarlo para eliminar espacio y caracteres especiales.

Nótese que el campo `image` está siendo validado con las reglas `image|dimensions:min_width=200,min_height=200` esto nos garantiza que el archivo que estamos recibiendo es efectivamente una imagen y que cumple con las dimensiones especificadas.

4. Para dar acceso público a los archivos que están en el directorio `storage/app/public`, se debe crear un enlace simbólico (acceso directo) hacia este directorio, dentro del directorio `public` que está en la raíz de nuestro proyecto. Para hacerlo, Laravel nos brinda un comando:

```
php artisan storage:link
```

TEST ON POSTMAN

DESCARGAR ARCHIVOS DESDE EL SERVIDOR

Una vez completados los pasos anteriores, podemos acceder al archivo almacenado en el servidor desde su URL:

```
http://localhost:8000/storage/articles/nombredelarchivo.jpeg
```

Si deseamos forzar al navegador a que descargue la imagen en lugar de mostrarla en pantalla, podemos realizar lo siguiente:

1. Crear una ruta para descargar la imagen del articulo, en el archivo `routes/api.php` añadimos esta línea ya sea en el grupo de rutas públicas o privadas. Si se la añade como ruta privada será necesario tener un token para descargar la imagen:

```
Route::get('articles/{article}/image', 'ArticleController@image');
```

2. En `ArticleController` añadimos el siguiente método:

```
public function image(Article $article)
{
  return response()->download(public_path(Storage::url($article->image)),
$article->title);
}
```

Si intentamos acceder desde Postman a la ruta `http://localhost:8001/api/articles/ARTICLE_ID/image` nos mostrará la imagen que hemos subido. Si accedemos a la misma ruta desde el navegador, nos mostrará la ventana para guardar el archivo en nuestro computador.

SEEDER CON IMÁGENES

Ahora que hemos incluido imágenes para nuestros artículos, debemos modificar el seeder para que pueda incluir imágenes ficticias. Faker nos ofrece un par de posibles soluciones:

1. `$faker->image`: Descarga una imagen aleatoria en la ruta que le indiquemos para cada artículo:

```
for ($i = 0; $i < 50; $i++) {
  $image_name = $faker->image('public/storage/articles', 400, 300, null,
false);
  Article::create([
    'title' => $faker->sentence,
    'body' => $faker->paragraph,
    'image' => 'articles/' . $image_name
  ]);
}
```

El método `$faker->image` está recibiendo como parámetros la ruta donde se descargará la imagen, el ancho, el alto, la categoría de la imagen y si el nombre de la imagen debe incluir la ruta completa. Para más detalles sobre este método podemos ver la documentación del método `$faker->image`: https://github.com/fzaninotto/Faker/blob/master/src/Faker/Provider/Image.php.

Esta sería la manera más realista de generar las imágenes, sin embargo, el servicio que utiliza (https://lorempixel.com/) tiene una respuesta lenta y eso hace que la generación de los seed tome demasiado tiempo. Una alternativa sería descargar una sola imagen y utilizarla para todos nuestros artículos.

2. `$faker->imageURL`: Este método genera una URL de lorempixel y lo incluye como valor del campo `image`, la desventaja es que cada vez que se presente un artículo, se presentará con una imagen diferente generada por lorempixel.

```
for ($i = 0; $i < 50; $i++) {
    Article::create([
        'title' => $faker->sentence,
        'body' => $faker->paragraph,
        'image' => $faker->imageUrl(400,300, null, false)
    ]);
}
```

TEST ON POSTMAN

PUSH TO GITHUB

AUTORIZACIÓN

Referencia: https://laravel.com/docs/6.x/authorization

Es importante aprender a diferenciar entre autenticación y autorización en un sistema. La **autenticación** tiene que ver con la existencia de un usuario en el sistema y sus credenciales de acceso. La **autorización** se refiere a que acciones puede realizar ese usuario dentro del sistema.

Laravel provee una forma simple de autorizar acciones del usuario contra un recurso dado. Como con la autenticación, el enfoque de Laravel para la autorización es simple, y hay dos maneras principales de autorizar acciones: **gates** y **policies** (puertas y políticas). Los gates y políticas son como rutas y controladores.

- **Gates**: proveen una manera simple, basada en funciones anónimas, para definir las reglas de autorización. Son más aplicables a acciones que no estén relacionadas a ningún modelo o recurso.
- **Políticas**: como los controladores, agrupan la lógica para un modelo o recurso en específico. Se utilizan cuando se desea autorizar una acción para un modelo o recurso en particular.

Para este curso implementaremos la autorización con políticas, pero es común mezclar gates y políticas.

POLITICAS (POLICIES)

Son clases que organizan la lógica de autorización para un modelo o recurso en particular. Por ejemplo, para el modelo `Article`, podemos generar una política `ArticlePolicy` para autorizar acciones de usuario como crear o actualizar artículos.

Para generar una política usamos el siguiente comando:

```
php artisan make:policy ArticlePolicy --model=Article
```

El argumento `--model=Article`, relaciona la política directamente con el modelo `Article` y genera los métodos necesarios que debemos completar con nuestra lógica de autorización.

Es recomendable seguir la convención de nombramiento de Laravel, esto quiere decir que, si los modelos están ubicados en el directorio `app`, las políticas deben estar en el directorio `app/Policies`. Además, las políticas deben coincidir con el nombre del modelo con el sufijo Policy. Por ejemplo, el modelo `Article` corresponderá a una clase `ArticlePolicy`. En caso de no seguir esta convención, se tendrá que registrar la política dentro del arreglo `$policies` del archivo `app/providers/AuthServiceProvider.php`.

Dentro de `ArticlePolicy` tenemos el método `update`, que determina si un usuario puede actualizar un artículo:

```
public function update(User $user, Article $article)
{
  return $user->id === $article->user_id;
}
```

Este método recibe el parámetro `$user` y `$article` los cuales están vinculados al usuario que realizó la petición y la instancia del modelo a actualizar. El método retorna un booleano que determina si el usuario tiene o no la autorización para realizar la acción. En el ejemplo, se autoriza la acción de actualizar si el `id` del usuario corresponde con el `user_id` del artículo. Asi, debemos completar la lógica de autorización para cada acción que los usuarios pueden realizar dentro del sistema.

ROLES Y PERMISOS

En la mayoría de los sistemas se cuenta con más de un tipo de usuarios, por ejemplo, podemos tener los usuarios comunes que tienen ciertas restricciones y además los usuarios administradores que tienen algunos privilegios. También en es común contar con un usuario súper administrador que no tiene ninguna restricción. Para poder manejar las restricciones de acuerdo con los tipos de usuarios, se manejan roles usuario dentro del sistema. Para este curso manejaremos 4 tipos de usuarios que pueden realizar las siguientes acciones:

Tabla 1 Roles y permisos

Acción	Usuario no autenticado	Usuario autenticado	Administrador	Súper administrador
Ver lista de artículos	x	x	x	x
Ver artículo individual		x	x	x
Añadir artículos		x	x	x
Editar sus propios artículos		x	x	x
Editar artículos de otros usuarios				x
Eliminar artículos			x	x
Ver comentarios		x	x	x
Añadir comentarios		x	x	x
Editar sus propios comentarios		x	x	x
Editar comentarios de otros usuarios				x
Eliminar comentarios			x	x
Ver lista de usuarios registrados			x	x
Ver usuario individual			x	x
Editar información de usuario				x
Eliminar usuario				x

Es importante notar la existencia de una jerarquía de los roles, es decir, los roles con más privilegios incluyen las acciones de los roles inferiores y otras acciones adicionales. Sin embargo, puede ser que no todos los roles sean manejados de manera jerárquica, es decir, que pueden existir roles con acciones mutuamente excluyentes entre si. Por ejemplo, podría existir un rol “Revisor” que podría tener los permisos de un usuario común y además podría tener permiso para editar los artículos de otros usuarios, pero podríamos mantener el rol “Administrador” sin el permiso para editar permisos de otros usuarios. Entonces vemos que los roles más privilegiados no necesariamente deben incluir todos los permisos de los roles inferiores.

Los roles y permisos pueden manejar una lógica muy sencilla o llegar a ser muy complejos. En este caso manejaremos una lógica medianamente sencilla.

Existen paquetes que nos permiten manejar los roles y permisos en Laravel, sin embargo, para mantener el sistema sencillo vamos a implementar el manejo de roles y permisos directamente con lo que Laravel nos proporciona.

Hay varias maneras de implementar los roles y permisos, vamos a implementar una manera sencilla y mencionar algunas variantes.

Para implementar los roles y permisos en nuestro sistema debemos seguir los siguientes pasos:

1. **Modificar el modelo** `User`**:** aumentar las siguientes constantes que contendrán los nombres de los roles y un método que nos permitirá conocer si un usuario ha sido asignado un rol, este método nos permitirá determinar si el usuario tiene permiso para realizar diferentes acciones en el sistema.

```
const ROLE_SUPERADMIN = 'ROLE_SUPERADMIN';
const ROLE_ADMIN = 'ROLE_ADMIN';
const ROLE_USER = 'ROLE_USER';

private const ROLES_HIERARCHY = [
    self::ROLE_SUPERADMIN => [self::ROLE_ADMIN, self::ROLE_USER],
    self::ROLE_ADMIN => [self::ROLE_USER],
    self::ROLE_USER => []
];

public function isGranted($role)
{
  return $role === $this->role || in_array($role,
self::ROLES_HIERARCHY[$this->role]);
}
```

   Hay que resaltar la manera en la que están definidos los roles, esto nos dice que un `SUPERADMIN` puede realizar las acciones que se le asignen y, además, todas las acciones de los `ADMIN` y los `USER`. Un `ADMIN` puede realizar todas las acciones asignadas a el directamente y además todas las acciones de un `USER`. En cambio, el rol `USER` solo podrá realizar las acciones que se le asignen directamente a el, no incluye las acciones de otros roles. Esta manera de implementarlo es fácil de entender y es flexible en la jerarquía, por ejemplo, podemos incluir un rol, `ROLE_REVIEWER` que incluya los permisos de `ROLE_USER` y además sea incluido por `ROLE_SUPERADMIN`, pero que no sea incluido por `ROLE_ADMIN`. Es decir, algo como lo siguiente:

```
const ROLE_SUPERADMIN = 'ROLE_SUPERADMIN';
const ROLE_ADMIN = 'ROLE_ADMIN';
const ROLE_REVIEWER = 'ROLE_REVIEWER'
const ROLE_USER = 'ROLE_USER';

private const ROLES_HIERARCHY = [
  self::ROLE_SUPERADMIN => [self::ROLE_ADMIN, ROLE_REVIEWER,
self::ROLE_USER],
  self::ROLE_ADMIN => [self::ROLE_USER],
  self::ROLE_REVIEWER => [self::ROLE_USER],
  self::ROLE_USER => []
];

public function isGranted($role)
{
  return $role === $this->role || in_array($role,
self::ROLES_HIERARCHY[$this->role]);
}
```

   Esto quiere decir que `ROLE_REVIEWER` podrá tener sus permisos, pero estos permisos no serán incluidos por `ROLE_ADMIN` y si serán incluidos por `ROLE_SUPERADMIN`.

Una variante de manera de manejar los roles es una en la que asumamos que se maneja una jerarquía estricta de roles, es decir, asumimos que los roles superiores incluyen a todos los inferiores. Esto se podría manejar así:

```
private const ROLES = [
  'ROLE_SUPERADMIN',
  'ROLE_ADMIN',
  'ROLE_USER'
];

public function isGranted($role)
{
  return array_search($role, self::ROLES) >= array_search($this->role,
self::ROLES);
}
```

Entonces la manera para determinar la jerarquía de roles sería conocer si el índice del rol del usuario es mayor que el índice del rol que se busca. Es decir, los roles en las primeras posiciones incluyen a los roles en las últimas posiciones.

Otra variante, quizás más eficiente sería establecer la jerarquía de los roles de la siguiente manera:

```
private const ROLES_HIERARCHY = [
  self::ROLE_SUPERADMIN => [self::ROLE_ADMIN],
  self::ROLE_ADMIN => [self::ROLE_USER],
  self::ROLE_USER => []
];
```

Es similar a la primera opción, sin embargo, no duplicamos los roles en la jerarquía. Se entiende que si el `ROLE_SUPERADMIN` incluye al `ROLE_ADMIN`, entonces también incluye a `ROLE_USER`. Sin embargo, el mecanismo para detectar si un rol está incluido en la jerarquía es más compleja:

```
public function isGranted($role)
{
  if ($role === $this->role) {
    return true;
  }

  return self::isRoleInHierarchy($role, self::ROLES_HIERARCHY[$this->role]);
}

private static function isRoleInHierarchy($role, $role_hierarchy)
{
  if (in_array($role, $role_hierarchy)) {
    return true;
  }

  foreach ($role_hierarchy as $role_included) {
    if(self::isRoleInHierarchy($role,self::ROLES_HIERARCHY[$role_included]))
    {
      return true;
    }
  }

  return false;
}
```

2. **Añadir la columna `role` en `users` mediante una migración:**

```
php artisan make:migration add_role_column_user
php artisan migrate
```

La migración tendrá lo siguiente:

```
class AddRoleColumnUser extends Migration
{
  public function up()
  {
    Schema::table('users', function (Blueprint $table) {
      $table->string('role')->default(\App\User::$ROLE_USER);
    });
  }

  public function down()
  {
    Schema::table('users', function (Blueprint $table) {
      $table->dropColumn('role');
    });
  }
}
```

3. **Modificar el** `ArticlePolicy`**:** La política ahora debe indicar, que role debe tener el usuario para poder realizar la acción. Entonces, debe verse así:

```
class ArticlePolicy
{
  use HandlesAuthorization;

  public function before($user, $ability)
  {
    if ($user->isGranted(User::ROLE_SUPERADMIN)) {
      return true;
    }
  }

  /**
   * Determine whether the user can view any articles.
   */
  public function viewAny(User $user)
  {
    return $user->isGranted(User::ROLE_ADMIN);
  }

  /**
   * Determine whether the user can view the article.
   */
  public function view(User $user, Article $article)
  {
    return $user->isGranted(User::ROLE_USER);
  }

  /**
   * Determine whether the user can create articles.
   */
  public function create(User $user)
  {
    return $user->isGranted(User::ROLE_USER);
  }

  /**
   * Determine whether the user can update the article.
   */
  public function update(User $user, Article $article)
  {
    return $user->isGranted(User::ROLE_USER) && $user->id === $article-
>user_id;
  }

  /**
   * Determine whether the user can delete the article.
   */
```

```
  public function delete(User $user, Article $article)
  {
    return $user->isGranted(User::ROLE_ADMIN);
  }
}
```

Hay que resaltar el método `before`, este es un hook que se ejecuta antes que cualquier política definida. Este se lo está utilizando para verificar si el usuario tiene el `ROLE_SUPERADMIN`, si es así entonces quiere decir que no tiene restricciones y se le permite realizar todas las acciones.

4. **Modificar el `ArticleController`:** Las acciones del controlador deben incluir el método `authorize` para verificar si el usuario está autorizado para ejecutar la acción. El primer parámetro que recibe este método debe coincidir con el nombre de alguna de las políticas establecidas en `ArticlePolicy`. El segundo parámetro es el nombre del modelo, o la instancia del modelo sobre la cual se va a realizar la acción. Por ejemplo:

```
public function index(Request $request)
{
  $this->authorize('viewAny', Article::class);
  return response()->json(ArticleResource::collection(Article::all()), 200);
}

public function show(Article $article)
{
  $this->authorize('view', $article);
  return response()->json(new ArticleResource($article), 200);
}
```

El método `index` no recibe una instancia del modelo `Article`, por lo tanto, el método `authorize` recibe el nombre del modelo sobre el cual se va a realizar la acción. En el caso del método `show`, se debe enviar la instancia del modelo sobre el cual se va a realizar la acción.

TEST ON POSTMAN

RELACIONES POLIMÓRFICAS

Referencia: https://laravel.com/docs/6.x/eloquent-relationships#polymorphic-relationships

Muchos sistemas que manejan roles de usuario se encuentran con el dilema de como manejar los diferentes tipos de usuarios que existen y los diferentes atributos que cada uno tiene. Esto podría terminar en una tabla `users` con muchos atributos y muchos de estos tendrán un valor `null` dependiendo del tipo de usuario al que corresponda el registro de la tabla. Es algo común pero no siempre recomendable debido a que la tabla podría crecer mucho y sea difícil de manejar y de entender.

Para aclarar este caso, vamos a suponer que en nuestro sistema los atributos de los usuarios van a depender del rol. Por ejemplo, podemos decir lo siguiente:

```
users:
  name
  lastname
  email
  password

admin:
```

```
  credential_number

writer
  editorial
  short_bio
```

Para casos como este, Eloquent proporciona las relaciones polimórficas. Estas permiten que un modelo pueda pertenecer a varios tipos de modelos con una sola asociación.

Igual que en las relaciones, estas pueden ser de uno a uno, uno a muchos o muchos a muchos. El ejemplo, que estamos proponiendo sería una relación de uno a uno ya que un `user` puede estar relacionado solo con un registro en `admin` o en `writer`. Para ver más detalles sobre las relaciones se puede referir a la documentación.

Para implementar esta relación polimórfica debemos seguir los siguientes pasos:

1. **Añadir las columnas `userable_id` y `userable_type` en `users`:**

```
php artisan make:migration add_userable_columns_users
```

La migración debe contener lo siguiente:

```
class AddUserableColumns extends Migration
{
    public function up()
    {
        Schema::table('users', function (Blueprint $table) {
            $table->integer('userable_id')->nullable();
            $table->string('userable_type')->nullable();
        });
    }

    public function down()
    {
        Schema::table('users', function (Blueprint $table) {
            $table->dropColumn('userable_id');
            $table->dropColumn('userable_type');
        });
    }
}
```

2. **Añadir los modelos para `admins` y `writers` con sus migraciones:**

```
php artisan make:model Admin -m
php artisan make:model Writer -m
```

Las migraciones serán las siguientes:
`create_admins_table.php`

```
class CreateAdminsTable extends Migration
{
    public function up()
    {
        Schema::create('admins', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('credential_number');
        });
    }

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

```
    }
}
```

`create_writers_table.php`

```
class CreateWritersTable extends Migration
{
    public function up()
    {
        Schema::create('writers', function (Blueprint $table) {
            $table->bigIncrements('id');
            $table->string('editorial');
            $table->text('short_bio');
        });
    }

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

3. **Establecer la relación entre los modelos:** Para esto debemos modificar los modelos `User`, `Admin` y `Writer`, así:

```
class Admin extends Model {
    public function user()
    {
        return $this->morphOne('App\User', 'userable');
    }
}


class Writer extends Model {
    public function user()
    {
        return $this->morphOne('App\User', 'userable');
    }
}


class User extends Authenticatable implements JWTSubject
{

    // [...]

    public function userable()
    {
        return $this->morphTo();
    }
}
```

Al completar estos pasos, ejecutamos la migración:

```
php artisan migrate
```

Ahora podremos acceder a la relación utilizando las propiedades dinámicas:

```
// Admin
$admin = App\Admin::find(1);
$user = $admin->user; // Devuelve la instancia de User


// Writer
```

```
$writer = App\Writer::find(1);
$user = $writer->user; // Devuelve la instancia de User


// User
$user = App\User::find(1);
$userable = $user->userable; // Devuelve la instancia de Admin o Writer
```

TEST ON POSTMAN

PUSH TO GITHUB

ENVÍO DE CORREOS

Referencia: https://laravel.com/docs/6.x/mail

Laravel proporciona una API limpia y simple sobre la popular biblioteca SwiftMailer (https://swiftmailer.symfony.com) con drivers para SMTP, Mailgun, Postmark, Amazon SES y `sendmail`, permitiéndote comenzar rápidamente a enviar correos a través de un servicio local o en la nube.

Los servicios basados en API como Mailgun o Postmark son más simples de implementar. Es recomendable utilizar alguno de estos servicios en lugar de SMTP. En este proyecto utilizaremos Mailgun.

CONFIGURACIÓN

1. Instalar la librería Guzzle HTTP:

```
composer require guzzlehttp/guzzle
```

2. Crear una cuenta en Mailgun: https://www.mailgun.com/
3. Editar las siguientes variables de entorno en el archivo `.env`:

```
MAIL_DRIVER=mailgun
MAILGUN_DOMAIN=tu-dominio-mailgun
MAILGUN_SECRET=tu-secreto-mailgun
```

   El dominio y secreto de mailgun se lo debe obtener desde la cuenta que se ha creado.

4. Verificar que el archivo `config/services.php` tiene las siguientes opciones:

```
'mailgun' => [
    'domain' => env('MAILGUN_DOMAIN'),
    'secret' => env('MAILGUN_SECRET'),
    'endpoint' => env('MAILGUN_ENDPOINT', 'api.mailgun.net'),
],
```

Para detalles de como configurar otros servicios para envío de correo diferentes a Mailgun, referirse al manual de Laravel.

CREAR EMAILS

Para crear un email se debe generar la clase con el siguiente comando:

```
php artisan make:mail NewComment
```

Esto generará la clase en `app/Mail/NewComment`.

Se debe completar la clase de la siguiente manera:

```
class NewComment extends Mailable
{
    use Queueable, SerializesModels;

    public $comment;

    public function __construct(Comment $comment)
    {
        $this->comment = $comment;
    }
```

```
    public function build()
    {

        return $this->view('emails.comments.new');

    }
}
```

El método `build` es donde se configura todo el correo a enviar. Aquí se puede llamar a los métodos `from`, `subject`, `view` y `attach`.

CONFIGURAR LA DIRECCIÓN DE ENVÍO DEL CORREO

Para configurar el campo `from` que irá en el correo tenemos dos maneras:

- Incluirlo directamente en el método `build`:

```
public function build()
{
    return $this->from('example@example.com')
                ->view('emails.comments.new');
}
```

- Utilizar la configuración global del campo `from`. Esto lo podemos configurar en el archivo de variables de entorno `.env`, y se lo utilizará en el archivo `config/mail.php`.

  `.env`

```
MAIL_FROM_ADDRESS=postmaster@blog.com
MAIL_FROM_NAME="${APP_NAME}"
```

  `config/mail.php`

```
'from' => [
    'address' => env('MAIL_FROM_ADDRESS', 'hello@example.com'),
    'name' => env('MAIL_FROM_NAME', 'Example'),
],
```

CONFIGURAR LA VISTA DEL CORREO

Dentro del método `build` podemos especificar la vista que utilizará un correo:

```
public function build()
{
    return $this->view('view.name');
}
```

Las vistas para los correos pueden utilizar el motor de plantillas Blade (https://laravel.com/docs/6.x/blade) lo cual nos facilita mucho la generación del contenido de los correos. Las plantillas de los correos deben estar en el directorio `resources/views/emails`. Para utilizar la plantilla del ejemplo, se debe crear el archivo dentro de `resources/views/emails/comments/new.blade.php`.

```
<html>
  <body>
    <h1>Hola! Tu artículo ha recibido nuevo comentario.</h1>
  </body>
</html>
```

También se pueden generar versiones de texto plano para los correos utilizando el método `text`. Se debe crear el archivo `resources/views/emails/comments/new_plain.blade.php`.

```
public function build()
{
    return $this->view('emails.comment.new')
                ->text('emails.comment.new_plain');
}
```

ENVIAR EL CORREO

Para enviar el correo, lo debemos hacer desde nuestro controlador. En este caso vamos a enviar un mail al autor del artículo siempre que alguien escriba un comentario. Lo hacemos de la siguiente manera:

```
public function store(Request $request, Article $article)
{
    $request->validate([
       'text' => 'required|string'
    ]);

    $comment = $article->comments()->save(new Comment($request->all()));
    Mail::to($article->user)->send(new NewComment($comment));
    return response()->json(new CommentResource($comment), 201);
}
```

También podemos configurar los campos cc y bcc:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->send(new NewComment($comment));
```

ENVÍO DE CORREOS DURANTE EL DESARROLLO

Mientras se encuentra en la etapa de desarrollo, los correos no deberían ser enviados a los destinatarios reales, para esto se recomienda utilizar una de las siguientes opciones:

MAILGUN SANDBOX

Si estamos trabajando con Mailgun, debemos configurar las variables de entorno `MAIL_USERNAME` y `MAIL_PASSWORD` con las credenciales de sandbox que nos proporciona Mailgun en nuestra cuenta. Este sandbox nos permite enviar correos hasta a 5 direcciones que hayan sido previamente añadidas como direcciones de prueba y estas hayan sido verificadas. En caso de que se intente enviar un correo a una dirección que no está añadida al sandbox, Mailgun nos devolverá un error.

CONFIGURAR UN "TO" UNIVERSAL

Se puede incluir una opción `to` en el archivo de configuración de correos `config.mail.php`.

```
'to' => [
    'address' => 'example@example.com',
    'name' => 'Example'
],
```

MAILTRAP

Otra opción es utilizar el servicio de Mailtrap (https://mailtrap.io/) que permite recibir los correos en una bandeja de entrada ficticia. Para esto se debe configurar las variables de entorno en el archivo `.env` correspondientes a las que nos proporcione Mailtrap.

PASAR VARIABLES A LA VISTA

Para pasar variables a la plantilla de nuestro correo, se pueden simplemente definir propiedades públicas en la clase del correo y estas serán automáticamente pasadas a la plantilla, por ejemplo, nuestra clase `CommentNew` se ve así:

```
class CommentNew extends Mailable
{
    use Queueable, SerializesModels;

    public $comment;

    public function __construct(Comment $comment)
    {
        $this->comment = $comment;
    }

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

Por lo que en la vista del correo podemos acceder a `comment` de la siguiente manera:

```
<html>
  <body>
    <h1>Hola! Tu artículo ha recibido nuevo comentario.</h1>
    <p>{{$comment->text}}</p>
  </body>
</html>
```

También se puede utilizar el método `with`:

```
public function build()
{
  return $this->view('emails.comments.new')
              ->with([
                  'commentText' => $this->comment->text,
                  'commentArticle' => $this->comment->article,
               ]);
}
```

ARCHIVOS ADJUNTOS

Para adjuntar archivos al correo podemos utilizar el método `attach`. Este recibe la ruta del archivo que se desea adjuntar. El segundo parámetro es un arreglo opcional y sirve para especificar el nombre con el que se adjuntará el archivo y el mime type.

```
public function build()
{
    return $this->view('emails.comments.new')
                ->attach('storage/archivo.pdf', [
                    'as' => 'name.pdf',
                    'mime' => 'application/pdf',
```

```
                ]);
}
```

Para incluir imágenes que deben ser mostradas en el contenido del correo debemos utilizar el método `embed` de la variable `$message` que es pasada automáticamente a todas las plantillas:

```
<body>
    Here is an image:

    <img src="{{ $message->embed($pathToImage) }}">
</body>
```

Por ejemplo, si la imagen está en nuestro storage entonces `$pathToImage` sería `storage/imagen.jpg`. Si queremos incluir la imagen asociada con nuestro artículo entonces `$pathToImage` sería `$comment->article->image`.

UTILIZAR MARKDOWN Y COMPONENTES DE LARAVEL PARA EL CONTENIDO DEL CORREO

Una manera más fácil y rápida de generar el contenido de los correos es utilizar markdown, para hacerlo debemos especificarlo al momento de generar el correo con el argumento `--markdown`:

```
php artisan make:mail NewComment --markdown=emails.comments.new
```

En lugar de llamar al método `view` en el método `build`, debemos llamar al método `markdown`:

```
public function build()
{
    return $this->from('example@example.com')
                ->markdown('emails.comments.new');
}
```

Ahora podemos escribir el contenido del correo así:

```
@component('mail::message')
# Hola!

## Tu artículo ha recibido nuevo comentario.

{{$comment->text}}

![Imagen del Artículo]({{asset('storage/' . $comment->article->image)}}
"Imagen")

@component('mail::button', ['url' => URL::to('/')])
Mira tu artículo aquí
@endcomponent

Gracias,<br>
{{ config('app.name') }}
@endcomponent
```

Los componentes disponibles son:

a. **Botón**

```
@component('mail::button', ['url' => $url, 'color' => 'success'])
View comment
@endcomponent
```

El color puede ser `success`, `primary` o `error`.

b. Panel

```
@component('mail::panel')
This is the panel content.
@endcomponent
```

Este componente renderiza el contenido en un color de fondo diferente al resto el mensaje por lo que sirve para atraer la atención.

c. Tabla

```
@component('mail::table')
| Laravel       | Table         | Example  |
| ------------- |:-------------:| --------:|
| Col 2 is      | Centered      | $10      |
| Col 3 is      | Right-Aligned | $20      |
@endcomponent
```

Nos permite construir una tabla HTML a partir de markdown. Notar los dos puntos (:) en las columnas 2 y 3 que nos permite centrar el texto o alinearlo a la derecha.

d. Promotion

```
@component('mail::promotion')
    This is the promotion content.
@endcomponent
```

Presenta el contenido dentro de una línea punteada.

e. Subcopy

```
@component('mail::subcopy')
    This is the subcopy content.
@endcomponent
```

Presenta el contenido con un texto ligeramente más pequeño.

Existen otros componentes como header, footer, layout que son incluidos automáticamente al utilizar el componente message. Si no utilizamos este componente, podemos incluir estos tres de manera manual.

PERSONALIZAR LOS ESTILOS DE LOS COMPONENTES

Para personalizar los estilos de los componentes debemos exportar los componentes a nuestro proyecto con el siguiente comando:

```
php artisan vendor:publish --tag=laravel-mail
```

Podemos modificar el archivo `default.css` que ahora se encuentra en el directorio `resources/views/vendor/mail/html/theme`.

También podemos generar un nuevo tema personalizado si añadimos un nuevo archivo css en `resources/views/vendor/mail/html/theme`. Además, debemos configurarlo en la opción `theme` del archivo `config/mail.php`.

```
'markdown' => [
    'theme' => 'default',

    'paths' => [
        resource_path('views/vendor/mail'),
```

```
    ],
],
```

Para personalizar el tema de un correo específico debemos definir la propiedad `$theme` con el nombre del tema de deseamos utilizar para ese correo en la clase mailable correspondiente.