



Sep-Dic 2023

CREACIÓN DE BASES DE DATOS.

Berrocal Chavez Alfredo

Ing. en Software

Grupo: 04-02





Contenido

1.	Creación de ORM(Object-Relational Mapping) relaciones avanzadas.....	3
2.	Relación uno a uno: Modelo User y Profile.....	5
3.	Relación uno a muchos: Modelo User Level	9
4.	Relación muchos a muchos: Modelo User y Group	15
5.	Relación uno a uno a través del Modelo User y Location	21
6.	Creación de migraciones y modelos, Category, Post y videos	26
1.	Creación del modelo Caterogy con su migración y factory.....	26
2.	Creación del modelo Post con su migración y factory	28
3.	Creación del modelo Video con su migración y factory.....	31
7.	Relación Polimórfica uno a muchos: El modelo users y comments.....	32
8.	Relación polimórfica uno a muchos: El modelo usuario e imágenes.....	35
1.	Creación del modelo Image con su migración y factory	35
9.	Relación Polimórfica muchos a muchos: El modelo users y tag	37
2.	Configuración de entidades	43
10.	Relación uno a muchos: Modelo User, Country, State y City	53
1.	Creación del modelo City	53
2.	Creación del modelo State	56
3.	Creación del modelo Country	59
11.	Relación uno a uno: Modelo User y Github_accounts.....	62



1. Creación de ORM(Object-Relational Mapping) relaciones avanzadas

Introducción.

En Laravel, el ORM a través de Eloquent **simplifica** la **interacción** entre **bases de datos relacionales** y lenguajes de programación orientados a objetos de una manera que resulta más accesible.

Al heredar de la clase '**Model**' y seguir cierta nomenclatura, se convierten las tablas en clases y se establece una **conexión automáticamente**.

Eloquent proporciona métodos para llevar a cabo operaciones **CRUD** y definir vínculos/relaciones entre modelos, lo que facilita el desarrollo al eliminar la necesidad de redactar consultas SQL directamente. Esto, a su vez, mejora la seguridad al prevenir inyecciones SQL y simplifica el mantenimiento gracias a un código legible y reutilizable.

En el contexto de Laravel, un ORM (Object-Relational Mapping) se convierte en una técnica que permite relacionar objetos de una aplicación con tablas de una base de datos relacional de manera completamente transparente. Laravel utiliza Eloquent como su ORM por defecto, el cual brinda una forma elegante y sencilla de interactuar con la base de datos mediante modelos y relaciones.

A continuación, te presento una tabla que describe las relaciones posibles en Eloquent:

Relación	Descripción	Siglas, o código
Uno a Uno	Una entidad está asociada con otra entidad de forma individual. Ejemplo: un usuario tiene un perfil.	HasOne
Uno a Muchos	Una entidad está asociada con múltiples instancias de otra entidad. Ejemplo: un autor tiene varios libros.	HasMany
Muchos a Uno	Múltiples instancias de una entidad están asociadas con una entidad en particular. Ejemplo: varios comentarios pertenecen a una publicación.	BelongsTo



Practica: Creación de bases de datos

Materia: Bases de datos

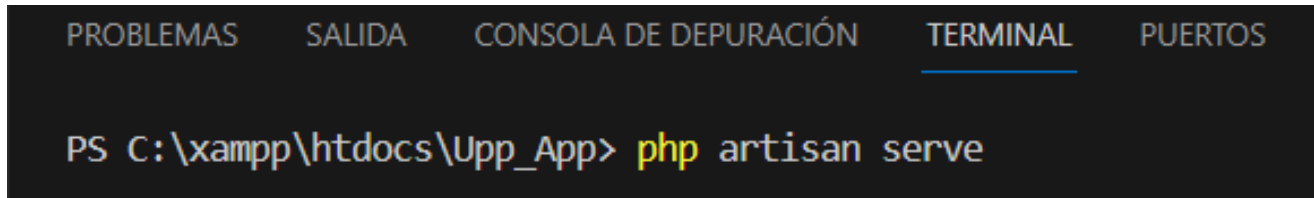
Elaborado: Alfredo Berrocal Chavez

Muchos a Muchos	Multiples instancias de una entidad esta n asociadas con multiples instancias de otra entidad. Ejemplo: varios usuarios tienen varios roles.	BelongsToMany
Polimórfica Uno a Uno	Una entidad está asociada con una de varias entidades diferentes. Ejemplo: un comentario puede estar asociado con una publicación o una foto.	MorphTo
Polimórfica Uno a Muchos	Una entidad está asociada con multiples instancias de varias entidades diferentes. Ejemplo: una etiqueta puede estar asociada con una publicación o una foto.	MorphTo
Polimórfica Muchos a Muchos	Multiples instancias de una entidad esta n asociadas con multiples instancias de varias entidades diferentes. Ejemplo: varias etiquetas pueden estar asociadas con varias publicaciones o fotos.	MorphToMany



2. Relación uno a uno: Modelo User y Profile

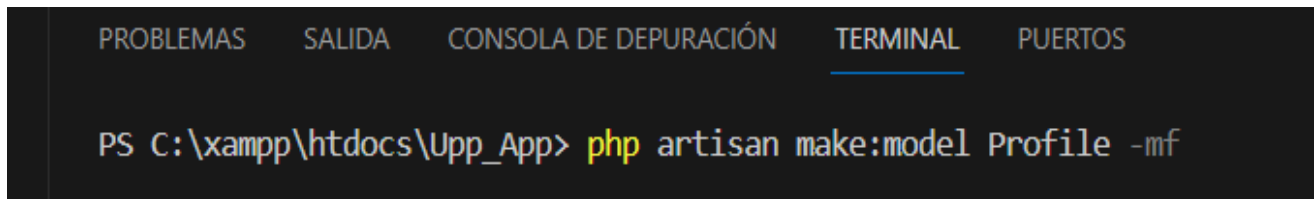
Iniciamos una terminal o consola de comandos en nuestro entorno de desarrollo de Laravel y ejecutamos los siguientes comandos para crear un modelo:



```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\xampp\htdocs\Upp_App> php artisan serve
```

Ilustración 1 comando levantar servidor



```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\xampp\htdocs\Upp_App> php artisan make:model Profile -mf
```

Ilustración 2 comando crear modelo

El comando "**php artisan make:model Profile -mf**" realiza las siguientes tareas:

"make:model Profile": Crea un nuevo modelo llamado "Profile". Laravel generará un archivo denominado "Profile.php" dentro de la carpeta "app" de tu proyecto. Este modelo se emplea para la interacción con los datos en la tabla correspondiente de la base de datos.

"-m": Genera una migración para el modelo "Profile". Laravel generará automáticamente un archivo de migración en la carpeta "**database/migrations**". La migración contendrá métodos para crear la tabla de la base de datos que corresponde al modelo "Profile". Posteriormente, puedes personalizar la migración según tus necesidades para agregar o modificar columnas.



"-f": Genera una factory para el modelo "Profile". Laravel creará un archivo de factory en la carpeta "**database/factories**". Las factories se emplean para generar datos de prueba para el modelo, lo que te permitirá crear registros ficticios de manera rápida y sencilla.

En resumen, el comando "php artisan make:model Profile -mf" te facilita la creación de un modelo "Profile", una migración para la tabla asociada y una factory para generar datos de prueba relacionados con dicho modelo. Esto te proporciona una estructura básica para comenzar a trabajar con el modelo "Profile" en tu proyecto de Laravel.

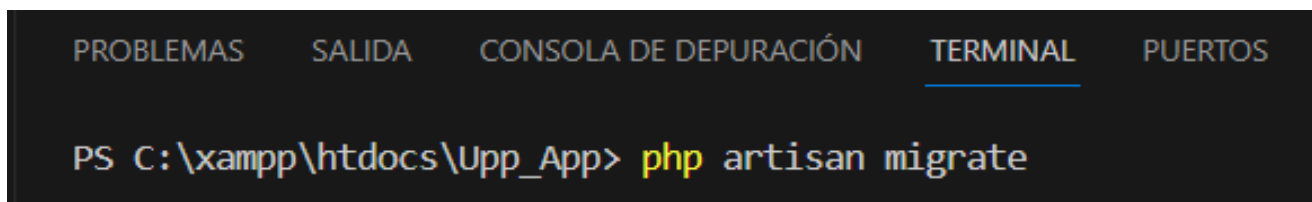


Ilustración 3 comando migración

"A continuación, vamos a establecer la relación entre el **modelo 'User'** y '**Profile**' dentro del archivo del modelo 'User'. Esta relación determina que un usuario está vinculado a un perfil utilizando la función 'hasOne()' proporcionada por Laravel."

Luego, se procede a dirigirse al archivo '**app\Models\User.php**' y se escribe el siguiente código.

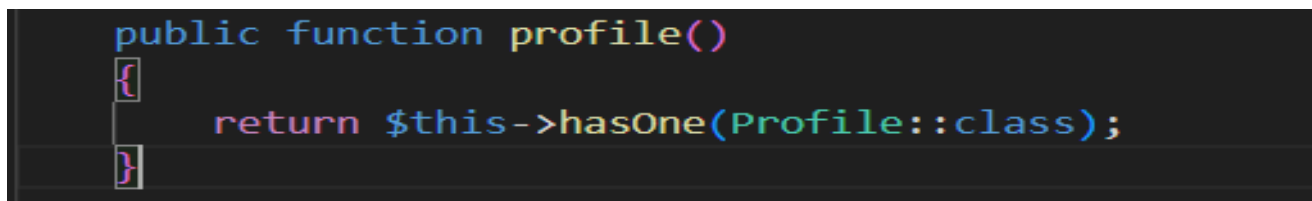


Ilustración 4 relación user profile

Este código define la relación entre el modelo User y el modelo Profile en Laravel. El método hasOne() establece que un usuario tiene un perfil asociado.



NOTA: un error muy frecuente es colocar “**return \$this**

>hasOne(App\Models\Profile::class);” ya que genera el siguiente error más adelante, al momento de intentar crear a los seeders:

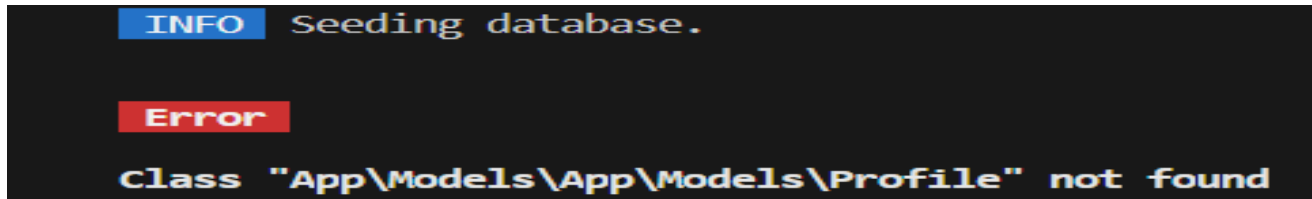


Ilustración 5 error1

Ahora iremos a configurar los campos para la migración en la tabla “**Profiles**” el cual se encuentran en “**database\Migrations\profiles_table.php**”

```
12 public function up(): void
13 {
14     Schema::create('profiles', function (Blueprint $table) {
15         $table->id();
16     }
17     $table->bigInteger('user_id')->unsigned(); #Codigo añadido
18
19     $table->string('instagram')->nullable();
20     $table->string('github')->nullable();
21     $table->string('web')->nullable();
22
23     $table->timestamps();
24
25     $table->foreign('user_id')->references('id')->on('users')
26         ->onDelete('cascade')
27         ->onUpdate('cascade');
28     });
29 }
```

Ilustración 6 código en profiles



\$table->bigInteger('user_id')->unsigned(); Este comando añade una columna denominada "user_id" de tipo bigInteger (entero grande) y sin signo a la tabla. Esta columna se utiliza para establecer la relación con el modelo User.

Luego, se añade una restricción de clave externa a la columna "user_id" utilizando el siguiente código:

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

Esta restricción de clave externa conecta la columna "user_id" con la columna "id" en la tabla "users". Esto establece una relación de clave externa entre las tablas "profiles" y "users", de modo que cualquier eliminación o actualización de un usuario en la tabla "users" se reflejará automáticamente en la tabla "profiles".

Una vez que hayas definido la estructura de la tabla "profiles" en la migración, puedes dirigirte a la consola de comandos de Laravel y ejecutar el siguiente comando.

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS  
  
PS C:\xampp\htdocs\Upp_App> php artisan migrate  
  
INFO Running migrations.  
  
2023_09_26_132410_create_profiles_table ..... 417ms DONE
```

Ilustración 7 comando migración

Y este sería el resultado en el diseñador de phpMyAdmin:

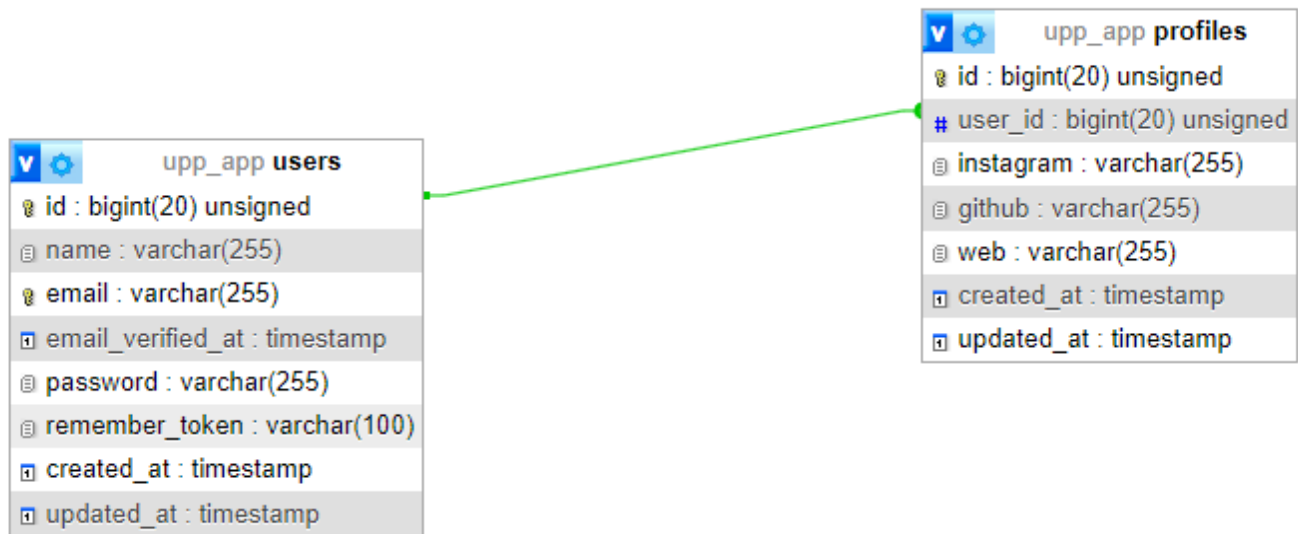


Ilustración 8 relación users profiles

3. Relación uno a muchos: Modelo User Level

Iniciamos una terminal o consola de comandos en nuestro entorno de desarrollo de Laravel y ejecutamos el siguiente comando para crear un modelo:

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\xampp\htdocs\Upp_App> php artisan make:model Level -mf

INFO Model [C:\xampp\htdocs\Upp_App\app\Models\Level.php] created successfully.
INFO Factory [C:\xampp\htdocs\Upp_App\database\factories\LevelFactory.php] created successfully.
INFO Migration [C:\xampp\htdocs\Upp_App\database\Migrations\2023_09_27_060031_create_levels_table.php] created successfully.
```

Ilustración 9 comando crear modelo

Cuando ejecutes este comando en la terminal o en la consola de comandos de Laravel, se generará automáticamente un modelo llamado "Level", junto con una migración y una factory asociadas.



A continuación, procederemos a definir la relación entre el modelo "User" y "Level" en el archivo del modelo "User". Esta relación establece que un "Level" tiene muchos usuarios asociados, utilizando el método "belongsTo()" proporcionado por Laravel.

A continuación, se muestra el código para establecer esta relación en el modelo "User" en el archivo "app\Models\User.php".

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

Ilustración 10 relacion en modelo user

En la migración agregamos el campo de "name" nos referimos al nombre del nivel como se muestra en la siguiente imagen:

```
public function up(): void  
{  
    Schema::create('levels', function (Blueprint $table) {  
        $table->id();  
        $table->string('name');  
        $table->timestamps();  
    });  
}
```

Ilustración 11 campo name



Iniciamos una terminal o consola de comandos en nuestro entorno de desarrollo de Laravel y ejecutamos el siguiente comando para crear un modelo:

```
PS C:\xampp\htdocs\Upp_App> php artisan make:migration add_level_id_at_users
```

```
INFO Migration [C:\xampp\htdocs\Upp_App\database\Migrations\2023_09_27_062533_add_level_id_at_users.p
```

Ilustración 12 crear modelo

Cuando ejecutes este comando, Laravel generará automáticamente un archivo de migración con un nombre único en el directorio "database/migrations". El nombre del archivo se basará en una marca de tiempo para asegurar su singularidad y se generará de manera similar a "20230614091234_add_level_id_at_users.php" (donde "20230614091234" representa la marca de tiempo actual).

Una vez que el comando se haya ejecutado con éxito, recibirás un mensaje en la consola que confirmará la creación exitosa del archivo de migración.

Una vez creada la migración agregaremos la siguiente relación:

```
Schema::table('users', function (Blueprint $table) {
```

```
    $table->bigInteger('level_id')->unsigned()->nullable()->after('id');
```

```
    $table->foreign('level_id')->references('id')->on('levels')->onDelete('set null')
```

```
    ->onUpdate('cascade');
```

```
});
```



```
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->bigInteger('level_id')->unsigned()->nullable()->after('id');

            $table->foreign('level_id')->references('id')->on('levels')->onDelete('set null')
            ->onUpdate('cascade');
        });
    }
}
```

Ilustración 13 relación

La función `Schema::table('users', function (Blueprint $table)` especifica que se realizara n cambios en la tabla 'users'.

`$table->bigInteger('level_id')->unsigned()->nullable()->after('id');` agrega una nueva columna llamada 'level_id' a la tabla 'users'. El tipo de datos se establece como `bigInteger` para almacenar valores enteros grandes.

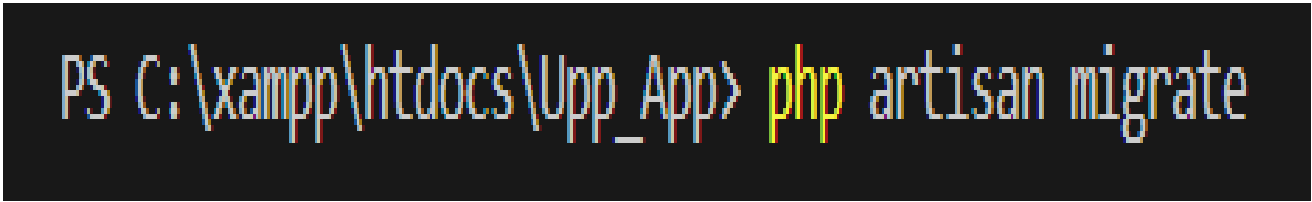
El método "`unsigned()`" asegura que únicamente se permitan valores positivos en la columna. La opción "`nullable()`" permite que la columna pueda contener valores nulos, lo que significa que no es obligatorio tener un valor de nivel asociado para todos los usuarios. El método "`after('id')`" especifica que la columna se posicionará después de la columna existente llamada 'id'.



Por otro lado, "`$table->foreign('level_id')->references('id')->on('levels')->onDelete('set null')->onUpdate('cascade');`" establece una relación de clave externa (foreign key) en la columna 'level_id' de la tabla 'users'. Esta línea indica que la columna 'level_id' hace referencia a la columna 'id' en la tabla 'levels'. La opción "`onDelete('set null')`" indica que si se elimina una fila en la tabla 'levels', se establecerá un valor nulo en la columna 'level_id' de la tabla 'users'. La opción "`onUpdate('cascade')`" indica que si se produce una actualización en la tabla 'levels', se actualizarán automáticamente todas las referencias en la tabla 'users'.

En resumen, este código de migración agrega la columna 'level_id' a la tabla 'users' y establece una relación de clave externa entre la columna 'level_id' y la tabla 'levels'. Esto permite asociar un nivel a cada usuario en la aplicación y mantener la integridad referencial entre las tablas.

Después abrimos la terminal de nuestro entorno de desarrollo y ejecutamos la migración:

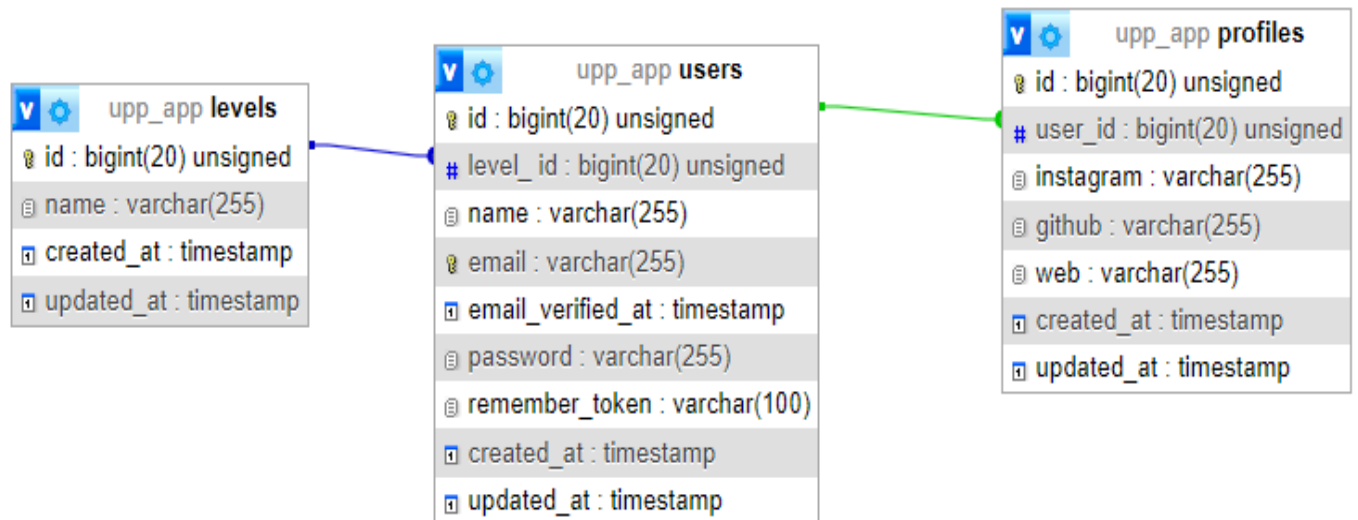


```
PS C:\xampp\htdocs\Upp_App> php artisan migrate
```

Ilustración 14 comando de migración



Y por último se mostrará el siguiente resultado en el diseñador de phpMyAdmin:





4. Relación muchos a muchos: Modelo User y Group

La relación de "muchos a muchos" es un tipo de vínculo en el cual varios registros de un modelo están conectados con varios registros de otro modelo.

En el caso de los modelos "**Group**" y "**User**", esto implica que un **grupo** puede tener múltiples **usuarios** y, a su vez, un **usuario** puede pertenecer a varios **grupos**. Para establecer esta relación, se utiliza una **tabla intermedia** que **actúa** como **intermediario** y guarda las **relaciones** entre los **grupos** y los **usuarios**. Esta configuración permite acceder de manera sencilla a los usuarios de un grupo y a los grupos a los que pertenece un usuario, lo que proporciona una relación flexible y versátil entre ambos modelos.

Para comenzar, abrimos una terminal o consola de comandos en nuestro entorno de desarrollo de Laravel y ejecutamos el siguiente comando para crear un modelo, junto con la migración y el factory asociados.

```
PS C:\xampp\htdocs\Upp_App> php artisan make:model Group -mf
```

Ilustración 15 comando crear modelo

Seguido de esto procederemos a crear la migración de la tabla intermedia con el siguiente comando dentro de la consola de nuestro entorno de desarrollo.

```
PS C:\xampp\htdocs\Upp_App> php artisan make:migration create_group_users_table
```

Ilustración 16 comando migración en tabla intermedia



Cuando ejecutes este comando, se generará un nuevo archivo de migración en la carpeta "database/migrations" de tu proyecto Laravel. El nombre del archivo de migración seguirá el formato de la marca de tiempo actual y el nombre especificado en el comando, que en este caso es "create_group_users_table". Dentro de este archivo de migración, tendrás la capacidad de definir la estructura de la tabla pivot, incluyendo las columnas necesarias para almacenar las relaciones entre los grupos y los usuarios.

Para añadir un campo de nombre a la migración de la tabla "groups", es necesario seguir estos pasos:

1. Abre la migración correspondiente a la tabla "groups". Puedes encontrarla en la carpeta "database/migrations" de tu proyecto Laravel.
2. Dentro del método "up()" de la migración, incluye el código necesario para crear el campo de nombre. Puedes utilizar el método "string('nombre')" para crear una columna de tipo cadena llamada "nombre". El código puede ser similar al siguiente:

```
public function up(): void
{
    Schema::create('groups', function (Blueprint $table) {
        $table->id();
        $table->string('nombre');
        $table->timestamps();
    });
}
```

Ilustración 17 crear campo nombre



Dentro de la migración llamada "create_group_users_table", tendrás la capacidad de especificar la estructura de la tabla pivot, lo que implica definir las columnas requeridas para almacenar las relaciones entre los grupos y los usuarios. Esto se ilustra en el siguiente fragmento de código.

```
public function up(): void
{
    Schema::create('group_users', function (Blueprint $table) {
        $table->id();
        $table->bigInteger('group_id')->unsigned();
        $table->bigInteger('user_id')->unsigned();
        $table->timestamps();

        $table->foreign('group_id')->references('id')->on('groups')->onDelete('cascade')->onUpdate('cascade');
        $table->foreign('user_id')->references('id')->on('users')->onDelete('cascade')->onUpdate('cascade');
    });
}
```

Ilustración 18 estructura de tabla pivot

En resumen, esta migración genera la tabla intermedia llamada "group_users". En esta tabla, se incluyen las siguientes columnas y configuraciones:

- "\$table->id();" agrega una columna de tipo ID que actúa como clave primaria de la tabla.
- "\$table->bigInteger('group_id')->unsigned();" añade una columna llamada "group_id" de tipo entero grande sin signo, destinada a almacenar el ID del grupo.
- "\$table->bigInteger('user_id')->unsigned();" incluye una columna llamada "user_id" de tipo entero grande sin signo, destinada a almacenar el ID del usuario.



- **"\$table->timestamps();" introduce las columnas "created_at" y "updated_at" para registrar las fechas de creación y actualización de los registros en la tabla "group_users".**

- **"\$table->foreign('group_id')->references('id')->on('groups')->onDelete('cascade')->onUpdate('cascade');" establece una clave externa en la columna "group_id", la cual hace referencia a la columna "id" en la tabla "groups". Esto garantiza que cuando se elimine o actualice un grupo, se realicen acciones en cascada en los registros relacionados en la tabla "group_users".**

- **"\$table->foreign('user_id')->references('id')->on('users')->onDelete('cascade')->onUpdate('cascade');" establece una clave externa en la columna "user_id", la cual hace referencia a la columna "id" en la tabla "users". Esto garantiza que cuando se elimine o actualice un usuario, se realicen acciones en cascada en los registros relacionados en la tabla "group_users".**

En resumen, esta migración crea la tabla intermedia "group_users" con columnas para almacenar las claves foráneas "group_id" y "user_id", estableciendo así una relación de muchos a muchos entre las tablas "users" y "groups".

Por último, abrimos la terminal de nuestro entorno de desarrollo y ejecutamos la migración:

```
PS C:\xampp\htdocs\Upp_App> php artisan migrate
```

Ilustración 19 comando migración

Y este sería el resultado final de la relación muchos a muchos:

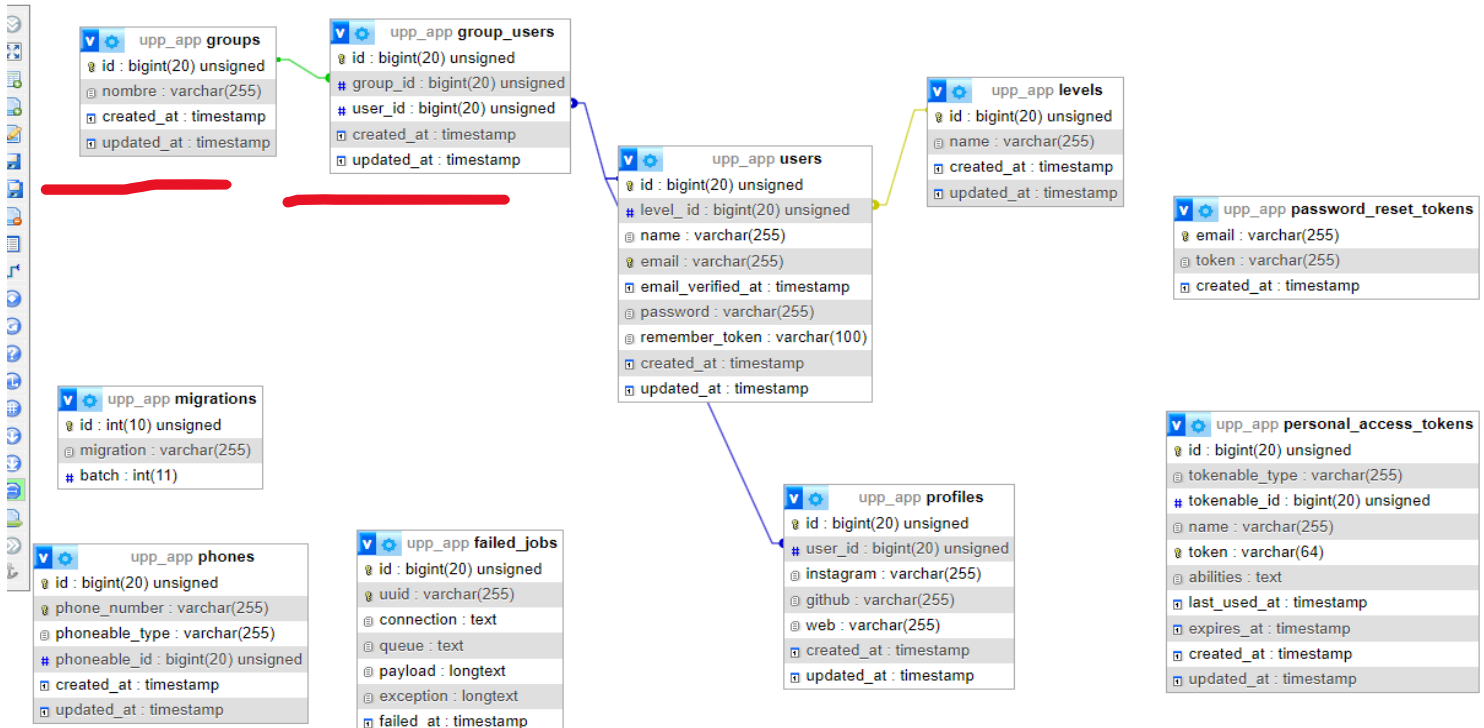


Ilustración 20 relación muchos a muchos

Ahora vamos a la configuración con el modelo: Object-Relational Mapping, así que nos dirigimos al modelo user.

```
public function groups()
{
    return $this->belongsToMany(Group::class)->withTimestamps();
}
```

Ilustración 21 configuración modelo user



Ahora seguiremos con la relación del modelo Group a User de la siguiente manera:

```
class Group extends Model
{
    use HasFactory;
    public function users()
    {
        return $this->belongsToMany(User::class)->withTimestamps();
    }
}
```

Ilustración 22 referencia group user



5. Relación uno a uno a través del Modelo User y Location

Una relación "uno a uno" a través de en Laravel se refiere a una conexión en la que un modelo está vinculado a otro modelo mediante la intermediación de un tercer modelo.

En el contexto de la relación entre "**User**" y "**Location**" a través de "**Profile**", esto significa que un **usuario** posee un perfil que, a su vez, está asociado a una ubicación específica. Esto se logra definiendo adecuadamente los métodos de relación en los modelos correspondientes.

Por ejemplo, en el modelo "User", se establecería el método "profile()" que utiliza "hasOne()" para establecer la relación con el modelo "Profile". En el modelo "Profile", se definiría el método "location()" que utiliza "hasOne()" para establecer la relación con el modelo "Location". Esta estructura permite un acceso sencillo a la ubicación de un usuario a través de su perfil.

Para comenzar, abrimos una terminal o consola de comandos en nuestro entorno de desarrollo de Laravel y ejecutamos el siguiente comando para crear un modelo, junto con la migración y el factory correspondientes.

```
PS C:\xampp\htdocs\Upp_App> php artisan make:model Location -mf  
  
INFO Model [C:\xampp\htdocs\Upp_App\app\Models\Location.php] created successfully.  
  
INFO Factory [C:\xampp\htdocs\Upp_App\database\factories\LocationFactory.php] created successfully.  
  
INFO Migration [C:\xampp\htdocs\Upp_App\database\Migrations\2023_09_28_031724_create_locations_table.php] created successfully.
```

Ilustración 23 comando crear modelo

Luego de ejecutar este comando, se generarán los siguientes archivos:

1. "Location.php": Este archivo corresponde al modelo "Location" y se creará en la carpeta "app/Models" de tu proyecto Laravel.
2. "create_locations_table.php": Esta migración se encarga de crear la tabla "locations" en la base de datos. El archivo de migración se generará en la carpeta "database/migrations" y contendrá los métodos "up()" y "down()" para definir la estructura de la tabla "locations".



3. "LocationFactory.php": Este archivo de factoría se empleará para generar datos de prueba relacionados con el modelo "Location". Se creará en la carpeta "database/factories".

Estos archivos son generados automáticamente para facilitar el desarrollo y la interacción con el modelo "Location" en tu proyecto Laravel.

Ahora iremos a agregar los campos para la migración de la tabla Location. Agregaremos los campos country, state, city y coordinates. Este último es para almacenar la ubicación GPS del usuario.

- Abre la migración correspondiente a la tabla "locations". Puedes encontrarla en la carpeta "database/migrations" de tu proyecto Laravel.

```
public function up(): void
{
    Schema::create('locations', function (Blueprint $table) {
        $table->id();

        $table->unsignedBigInteger('profile_id');

        $table->string('country')->nullable();
        $table->string('state')->nullable();
        $table->string('city')->nullable();
        $table->point('coordinates');

        $table->foreign('profile_id')->references('id')->on('profiles')
            ->onDelete('cascade')
            ->onUpdate('cascade');

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

Ilustración 24 campos de locations



1. `"$table->id()"`: Este método genera automáticamente una columna de tipo "id" que funciona como la clave primaria de la tabla "locations".
2. `"$table->bigInteger('profile_id')->unsigned()"`: Esta instrucción añade una columna llamada "profile_id" de tipo "bigInteger" y se emplea como clave externa para establecer la relación entre la tabla "locations" y la tabla "profiles".
3. `"$table->string('country')->nullable(), $table->string('state')->nullable(), $table->string('city')->nullable()"`: Estas líneas crean columnas de tipo "string" destinadas a almacenar información sobre el país, estado y ciudad de la ubicación. La utilización de "nullable()" indica que estos campos pueden tener valores nulos en la base de datos.
4. `"$table->point('coordinates')"`: Esta sentencia crea una columna llamada "coordinates" de tipo "point" que se usa para guardar las coordenadas geográficas de la ubicación GPS.
5. `"$table->foreign('profile_id')->references('id')->on('profiles')->onDelete('cascade')->onUpdate('cascade')"` establece una relación de clave externa entre la columna "profile_id" de la tabla "locations" y la columna "id" de la tabla "profiles". La opción "onDelete('cascade')->onUpdate('cascade')" especifica que si se elimina o actualiza un perfil, se realizarán acciones en cascada en los registros relacionados en la tabla "locations".
6. `"$table->timestamps()"` crea automáticamente las columnas "created_at" y "updated_at" para registrar la fecha y hora de creación y actualización de los registros.

En resumen, este código genera la tabla "locations" que incluye columnas para el perfil asociado, país, estado, ciudad y coordenadas geográficas. Además, establece una relación de clave externa con la tabla "profiles" y agrega marcas de tiempo para registrar la información de creación y actualización de los registros.



Luego, para llevar a cabo esta migración, abrimos la terminal en nuestro entorno de desarrollo y ejecutamos el comando correspondiente:

```
PS C:\xampp\htdocs\Upp_App> php artisan migrate
```

INFO Running migrations.

Ilustración 25 comando migración

Por último, el resultado final uno a uno a través de un usuario, tiene una location y a través de perfiles

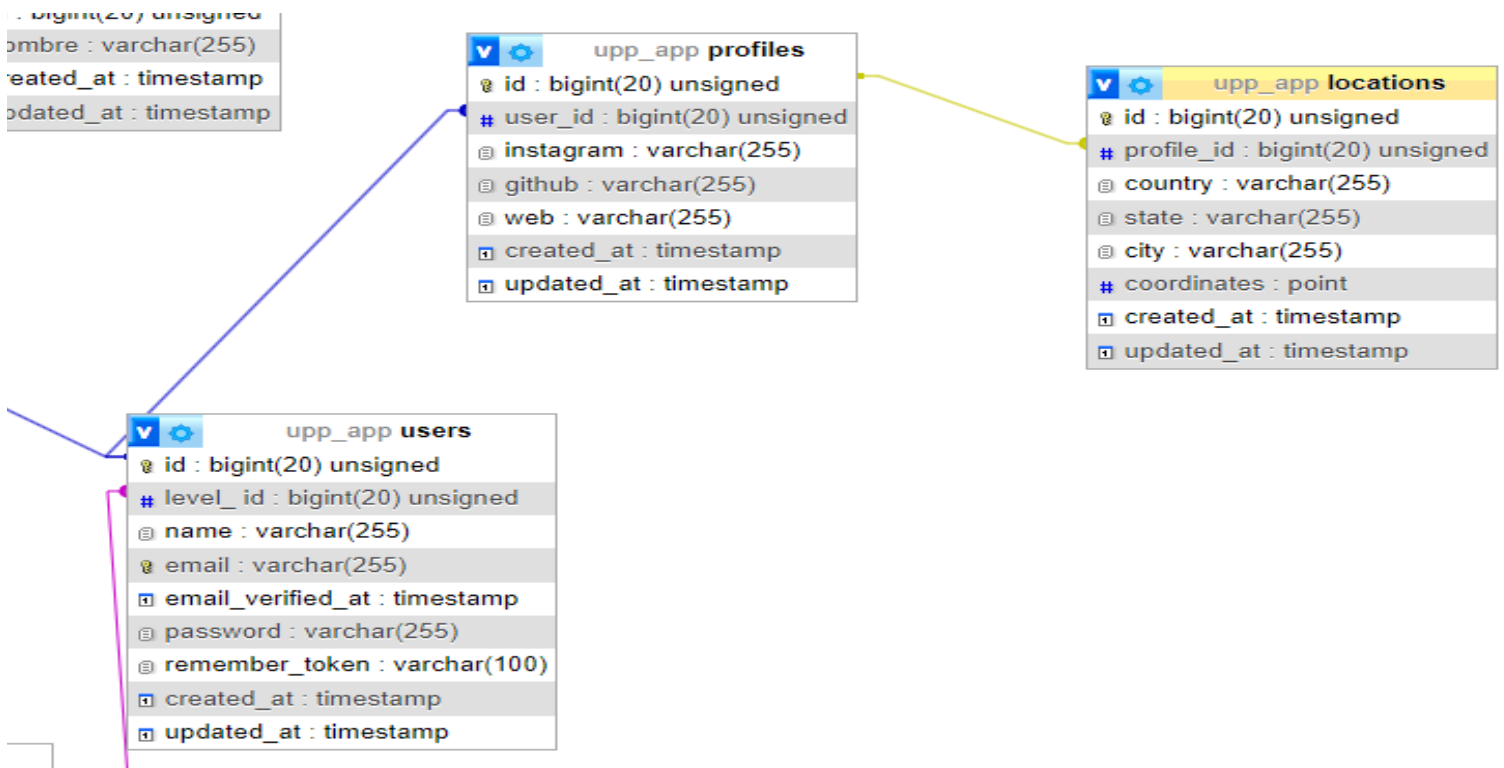


Ilustración 26 relación location y profile



Ahora vamos a la configuración con el modelo: Object-Relational Mapping. Nos dirigimos al modelo user:

```
public function location()  
{  
    return $this->hasOneThrough(Location::class, Profile::class);  
}
```

Ilustración 27 location en modelo user

Este método establece una relación uno a uno entre el modelo de **usuario** y el modelo de **ubicación** a través del modelo de **perfil**. Significa que un **usuario tiene** una **ubicación** asociada a través de su **perfil**.

Al emplear "**hasOneThrough()**" en Laravel, el sistema establece automáticamente las conexiones entre los modelos basándose en las claves foráneas y las relaciones definidas en los modelos respectivos.

Ahora, configuramos el modelo "**profile**" para establecer la relación con la tabla "**location**" de la siguiente manera:

```
public function location()  
{  
    return $this->hasOne(Location::class);  
}
```

Ilustración 28 relación con tabla location



6. Creación de migraciones y modelos, Category, Post y videos

Es una práctica común comenzar por las entidades **principales** o más **generales** antes de avanzar hacia las entidades **secundarias** o más **específicas**.

En este caso, podríamos comenzar con "**category**" como una elección adecuada, seguida de "**post**" y "**video**", que podrían estar **relacionados** con "category". Posteriormente, "**comment**" podría agregarse como una **entidad vinculada** a "**post**" o "**video**".

Es fundamental recordar que al crear las migraciones y definir las relaciones en los modelos, es necesario considerar las restricciones de clave externa y garantizar las relaciones adecuadas entre las tablas. Además, es esencial asegurarse de que la secuencia en la que se ejecutan las migraciones coincida con las dependencias entre las tablas.

En términos generales, la secuencia de creación de modelos y migraciones depende del diseño de tu base de datos y de cómo deseas organizar tus entidades. Asegúrate de analizar y planificar correctamente las relaciones entre las tablas para evitar problemas en el futuro.

Para comenzar, abre una terminal o consola de comandos en tu entorno de desarrollo de Laravel y ejecuta el siguiente comando para crear un modelo, junto con la migración y el factory correspondientes:

1. Creación del modelo **Category** con su migración y factory

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\xampp\htdocs\Upp_App> php artisan make:model Category -mf
```

Ilustración 29 comando creación de modelo

Una vez ejecutado el comando, Laravel generará automáticamente tres archivos:



- **Modelo:** Se generará un archivo llamado "Category.php" en la carpeta "app/Models" de tu proyecto Laravel. Este archivo contendrá la estructura básica del modelo "Category", incluyendo métodos y propiedades esenciales.
- **Migración:** Se creará un archivo de migración en la carpeta "database/migrations" de tu proyecto Laravel. El nombre del archivo incluirá una marca de tiempo y el nombre de la tabla, que en este caso sería "create_categories_table". Puedes abrir este archivo y agregar los campos y configuraciones necesarios para la tabla "categories".
- **Factory:** Se generará un archivo llamado "CategoryFactory.php" en la carpeta "database/factories" de tu proyecto Laravel. Este archivo contendrá la definición de una factoría para la generación de datos de prueba relacionados con el modelo "Category".

Para agregar el campo "name" a la tabla "categories" en tu entorno de desarrollo, sigue estos pasos:



1. Dirígete al directorio "database/migrations" de tu proyecto Laravel.
2. Busca el archivo de migración relacionado con la tabla "categories". El nombre del archivo debería seguir un formato específico.
3. Abre el archivo de migración en un editor de código.
4. En el método "up()", añade una nueva instrucción para crear el campo "name".

```
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('categories', function (Blueprint $table) {
            $table->id();
            $table->string('name');
            $table->timestamps();
        });
    }
}
```

Ilustración 30 creación de campo categories

De nueva cuenta iremos a la consola para crear el modelo del post:

```
PS C:\xampp\htdocs\Upp_App> php artisan make:model Post -mf
```

Ilustración 31 comando crear modelo



Además, añadiremos el controlador al "post" ya que lo necesitaremos en el futuro.

Para crear un controlador en Laravel con todos los recursos necesarios, puedes emplear el comando "php artisan make:controller" junto con la opción "--resource". A continuación, te presento el comando completo que utilizarías para generar un controlador con todos los recursos para el modelo "Post":

```
PS C:\xampp\htdocs\Upp_App> php artisan make:controller PostController --resource
```

Ilustración 32 comando crear controlador

Utilizando este comando, Laravel creará un controlador denominado "PostController" que se ubicará en la carpeta "app/Http/Controllers" de tu proyecto. Además, este controlador se generará con todos los métodos y acciones necesarios para los recursos RESTful, que incluyen "index", "create", "store", "show", "edit", "update" y "destroy".

Para incorporar el campo "message" a la tabla "categories" en tu entorno de desarrollo, sigue estos pasos:

```
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('posts', function (Blueprint $table) {
            $table->id();
            $table->bigInteger('category_id')->unsigned()->nullable();
            $table->string('message');

            //Restricción de clave foránea para la columna 'category_id'
            $table->foreign('category_id')->references('id')->on('categories')
                ->onDelete('cascade') //Eliminación en cascada si la categoría se elimina
                ->onUpdate('cascade');//Actualización en cascada si el id de la categoría cambia

            //Restricción de clave foránea utilizando foreignId para la columna 'user_id'
            $table->foreign('user_id')->references('id')->on('users')
                ->onDelete('cascade') //Eliminación en cascada si el usuario se elimina
                ->onUpdate('cascade');//Actualización en cascada si el id del usuario cambia

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

Ilustración 33 creación de campos den migración pots



La migración proporcionada genera una tabla denominada "posts" que está vinculada a las tablas "categories" y "users". Para establecer esta relación, se incluyen las siguientes columnas:

1. La columna "category_id": Se trata de un número entero grande (bigInteger) que hace referencia a la clave primaria (id) de la tabla "categories". Esta columna permite almacenar el ID.
2. La columna "message": Consiste en una cadena de texto (string) destinada a almacenar el contenido del mensaje del post.

Además de las columnas mencionadas, se definen restricciones de clave externa para establecer la relación entre las tablas de la siguiente manera:

- Para la columna "category_id", se emplea el método "foreign()" para especificar la restricción de clave externa. Esta restricción dicta que el valor de "category_id" debe coincidir con un valor válido en la columna "id" de la tabla "categories". Se agrega la opción "onDelete('cascade')" para indicar que, si se elimina una categoría, todos los posts relacionados también se eliminarán en cascada. De manera análoga, "onUpdate('cascade')" establece que si el ID de una categoría cambia, se actualizará en cascada en los posts relacionados.
- Para la columna "user_id", se utiliza el método "foreignId()" para definir la restricción de clave externa. La sintaxis "foreignId('user_id')->references('id')->on('users')" especifica que "user_id" debe coincidir con un valor válido en la columna "id" de la tabla "users". Al igual que en la restricción anterior, se utilizan "onDelete('cascade')" y "onUpdate('cascade')" para establecer la eliminación y actualización en cascada.



3. Creación del modelo **Video** con su migración y factory

Ahora continuaremos con la creación de video con ayuda del siguiente comando:

```
PS C:\xampp\htdocs\Upp_App> php artisan make:model Video -mf
```

Al ejecutar este comando en la consola, Laravel generara automáticamente el modelo, la migración y la factoría correspondientes al modelo "**Video**". La consola mostrara un mensaje indicando que el modelo ha sido creado exitosamente, seguido de los mensajes que confirma

Nos dirigimos a la migración para agregar lo siguiente:

```
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('videos', function (Blueprint $table) {
            $table->id();
            $table->bigInteger('category_id')->unsigned()->nullable();

            $table->string('message');
            $table->string('video_path');//Campo para almacenar la ruta o nombre del archivo

            //Restricciones de clave foranea
            $table->foreign('category_id')->references('id')->on('categories')
                ->onDelete('cascade') //Eliminación en cascada si la categoría se elimina
                ->onUpdate('cascade');//Actualización en cascada si el id de la categoría cambia

            $table->foreignId('user_id')->references('id')->on('users')
                ->onDelete('cascade') //Eliminación en cascada si el usuario se elimina
                ->onUpdate('cascade');//Actualización en cascada si el id del usuario cambia

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

Ilustración 34 configuración de migración de videos



7. Relación Polimórfica uno a muchos: El modelo users y comments

Introducción:

La relación **uno a muchos polimórfica** entre los modelos "**Usuario**" y "**Comentario**" implica que un **usuario** puede tener **múltiples comentarios** asociados, y estos comentarios pueden estar **vinculados** a diferentes tipos de **entidades**.

Una relación polimórfica entre los modelos "**Usuario**" y "**Comentario**" permite que un modelo establezca una relación con múltiples tipos de modelos diferentes. En este caso, la relación polimórfica se establece mediante el uso del método "**morphs()**" al crear la tabla de comentarios.

En términos prácticos, la relación polimórfica permite que un **comentario** pueda estar **asociado** a diferentes tipos de **entidades**, como publicaciones, imágenes, videos, etc.

En lugar de crear una columna separada para cada tipo de entidad (por ejemplo, "post_id", "image_id", "video_id"), se utiliza la columna "**commentable**" como un campo polimórfico que **almacena** tanto la **clave foránea** como el tipo de **entidad relacionada**.

Esto permite que un comentario esté asociado a diferentes entidades sin necesidad de crear tablas adicionales o modificar la estructura de la tabla de comentarios. Al utilizar la relación polimórfica, Laravel proporciona métodos convenientes para acceder y administrar estas relaciones de manera transparente y eficiente.

Con esta introducción, procederemos con la parte práctica. Abre una terminal en tu entorno de desarrollo y crea la tabla "Comment" junto con la migración y la factoría correspondientes, como lo hemos estado realizando anteriormente.

```
PS C:\xampp\htdocs\Upp_App> php artisan make:model Comment -mf
```

Ilustración 35 comando creación de modelo



Este comando nos ayuda a crear el modelo “Comment” que necesitamos junto con su migración y factory correspondientes en Laravel.

Posteriormente nos dirigimos a la migración para realizar la relación polimórfica de los comentarios:

```
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('comments', function (Blueprint $table) {
            $table->id();

            $table->bigInteger('user_id')->unsigned();
            $table->string('body');

            $table->morphs('commentable');
            $table->timestamps();

            $table->foreign('user_id')->references('id')->on('users')
                ->onDelete('cascade')
                ->onUpdate('cascade');
        });
    }
}
```

Ilustración 36 relación polimorfa de comentarios



1. La tabla "comments" se genera mediante el uso del método `Schema::create()` en la migración.
2. Se incorpora la columna "user_id" como una clave foránea de tipo `bigInteger`, la cual representa el ID del usuario que efectuó el comentario.
3. Se incluye la columna "body" como una cadena de texto (string), destinada a guardar el contenido del comentario.
4. Se emplea el método `"morphs('commentable')"` para crear dos campos, "commentable_type" y "commentable_id", que facilitan el establecimiento de una relación polimórfica con otras entidades. Estos campos se utilizan para almacenar tanto el tipo de entidad relacionada como su identificador.
5. Se adicionan las columnas `"timestamps()"` para insertar automáticamente los campos "created_at" y "updated_at" en la tabla "comments".
6. Se establece una restricción de clave foránea para la columna "user_id" mediante el uso del método `"foreign()"`. Esta restricción especifica que la columna "user_id" debe hacer referencia a la columna "id" en la tabla "users".
7. Se configura la acción `"onDelete('cascade')"` para la restricción de clave foránea de "user_id", lo que implica que si se elimina un usuario, todos los comentarios asociados a ese usuario también se eliminarán en cascada.
8. Se configura la acción `"onUpdate('cascade')"` para la restricción de clave foránea de "user_id", lo que significa que si el ID de un usuario se actualiza, los comentarios relacionados con ese usuario también se actualizarán en cascada.



8. Relación polimórfica uno a muchos: El modelo usuario e imágenes

1. Creación del modelo **Image** con su migración y factory

```
PS C:\xampp\htdocs\Upp_App> php artisan make:model Image -mf
```

Ilustración 37 comando para crear modelo junto con factory

Este comando generará los siguientes archivos:

1. `app/Models/Image.php`: El archivo del modelo correspondiente al modelo "Image", se ubicará en el directorio `app/Models`.
2. `database/migrations/YYYY_MM_DD_HH:mm:ss_create_images_table.php`: El archivo de migración destinado a crear la tabla "images" en la base de datos. La marca de tiempo en el nombre del archivo representa la fecha y hora actual.
3. `database/factories/ImageFactory.php`: El archivo de factoría diseñado para generar datos ficticios relacionados con el modelo "Image".

Es importante señalar que, de manera predeterminada, Laravel sigue el estándar de autocarga PSR-4, lo que significa que los modelos se almacenan en el directorio `app/Models`. No obstante, si prefieres ubicar tus modelos directamente en el directorio `app`, puedes omitir el subdirectorio "Models" al utilizar el comando.

Ahora, procederemos a configurar la migración para la imagen:

```
return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create('images', function (Blueprint $table) {
            $table->id();
            $table->string('url');
            $table->morphs('imageable');
            $table->timestamps();
        });
    }
}
```

Ilustración 38 configuración de la migración de imagen

1. Se genera la tabla "images" empleando el método Schema::create() en la migración.
2. Se incorpora la columna "id" como una clave primaria autoincremental (id()).
3. Se añade la columna "url" como una cadena de texto (string), destinada a guardar la URL de la imagen.
4. Se utiliza el método "morphs('imageable')" para crear dos campos, "imageable_type" y "imageable_id", que facilitan el establecimiento de una relación polimórfica con otras entidades.

Estos campos se emplearán para almacenar tanto el tipo de entidad relacionada como su identificador. Además, se incorporan las columnas "timestamps()" con el fin de añadir automáticamente los campos "created_at" y "updated_at" a la tabla "images".



9. Relación Polimórfica muchos a muchos: El modelo users y tag

Introducción:

En el contexto de una relación polimórfica entre el modelo "Usuario" y el modelo "tag" en Laravel, se puede emplear una relación polimórfica de **muchos a muchos**. Esto implica que un **usuario** puede tener múltiples **etiquetas** asociadas, y estas **etiquetas** también pueden estar **vinculadas** a otras **entidades diversas**.

Para establecer esta relación, se utilizarán los métodos "**morphToMany()**" y "**morphedByMany()**", pero antes de ello habrá que crear el modelo, de la siguiente manera:

```
PS C:\xampp\htdocs\Upp_App> php artisan make:model Tag -mf
INFO Model [C:\xampp\htdocs\Upp_App\app\Models\Tag.php] created successfully.
INFO Factory [C:\xampp\htdocs\Upp_App\database\factories\TagFactory.php] created successfully.
INFO Migration [C:\xampp\htdocs\Upp_App\database\Migrations\2023_09_29_020831_create_tags_table.php] created successfully.
```

Ilustración 39 comando para crear modelo

Este comando generará los siguientes archivos:

1. `app/Models/Tag.php`: El archivo del modelo correspondiente al modelo "Tag", ubicado en el directorio `app/Models`.
2. `database/migrations/YYYY_MM_DD_HH:mm:ss_create_tags_table.php`: El archivo de migración destinado a crear la tabla "tags" en la base de datos. La marca de tiempo en el nombre del archivo representa la fecha y hora actual.
3. `database/factories/TagFactory.php`: El archivo de factoría diseñado para generar datos ficticios relacionados con el modelo "Tag".



Es importante recordar que la opción -mf se utiliza para generar tanto la migración como la factory en un solo comando.

Con esto realizado, nos dirigiremos a el modelo "Usuario", allí es donde se definirá el método "morphToMany()" especificando el modelo objetivo (en este caso, "Etiqueta") y la tabla intermedia que almacenará las relaciones polimórficas.

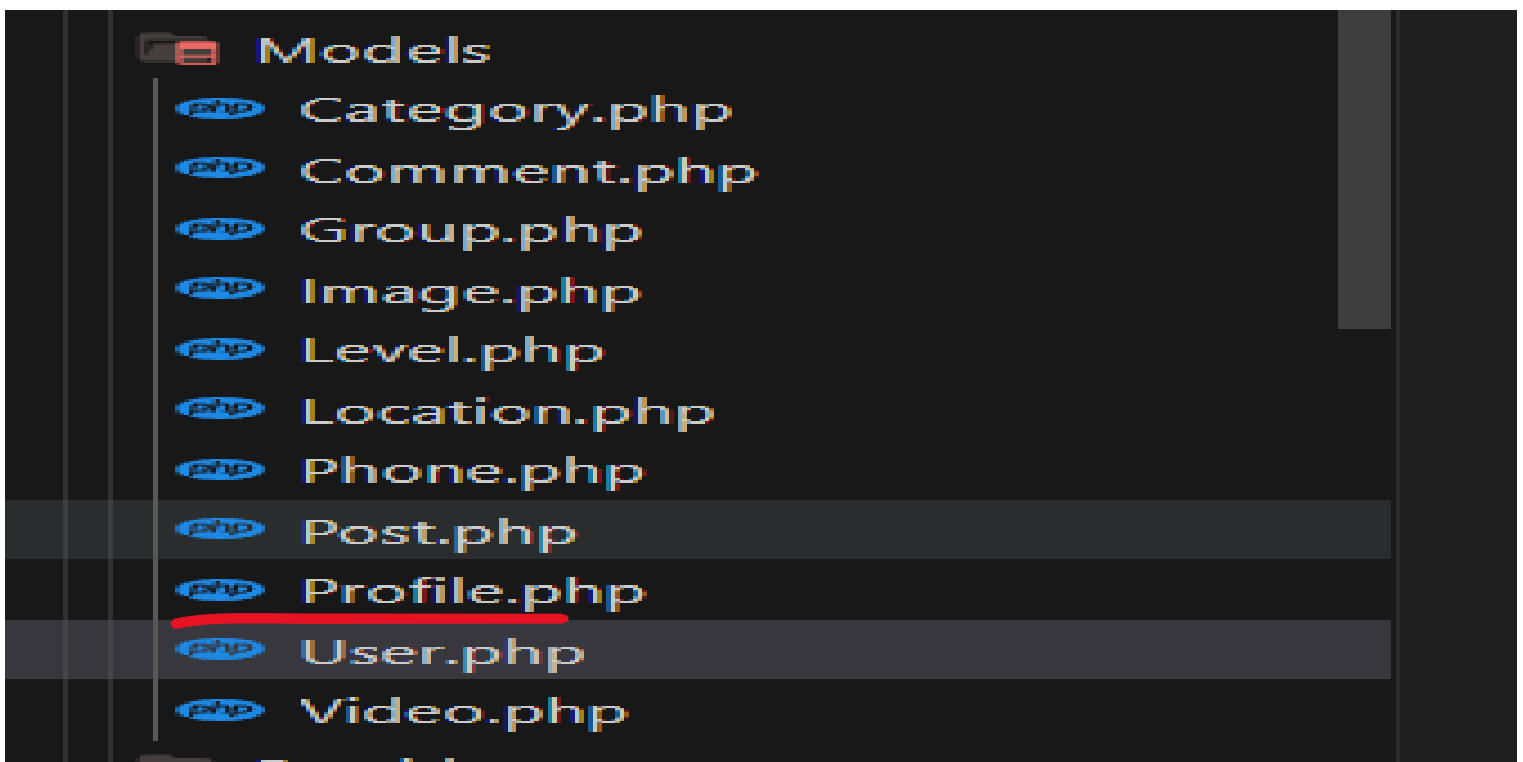


Ilustración 40 ubicación



```
public function tags()  
{  
    return $this->morphToMany(Tags::class, 'taggable');  
}
```

Ilustración 41 método morphedMany en usuario

En el modelo “Etiqueta”, se utiliza el método ‘**morphedByMany()**’ para definir la relación polimórfica inversa. Por ejemplo:

```
class Tag extends Model  
{  
    use HasFactory;  
  
    public function users()  
    {  
        return $this->morphedByMany(Usuarios::class, 'taggable');  
    }  
}
```

Ilustración 42 método morphedMany en tag



A la migración tags_table.php le agregamos el campo "name" como se muestra a continuación.

```
6
7 return new class extends Migration
8 {
9     /**
10      * Run the migrations.
11      */
12     public function up(): void
13     {
14         Schema::create('taggables', function (Blueprint $table) {
15             $table->id();
16             $table->string('name');
17             $table->timestamps();
18         });
19     }
20
21     /**
22      * Reverse the migrations.
23      */
24     public function down(): void
25     {
26         Schema::dropIfExists('taggables');
27     }
28 }
```

Ilustración 43 campo name en migración tags

Para poder realizar la relación de muchos a muchos necesitamos una tabla intermedia por la cual solo crearemos la migración con el siguiente comando:

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\xampp\htdocs\Upp_App> php artisan make:migration create_taggables_table
```

Ilustración 44 creación de migración, tabla intermedia

Este comando producirá un archivo de migración nombrado "YYYY_MM_DD_HH:mm:ss_create_taggables_table.php", el cual se ubicará en el directorio



"database/migrations". La marca de tiempo en el nombre del archivo refleja la fecha y hora actuales.

A continuación, procederemos a establecer la relación de muchos a muchos en las migraciones de la tabla que acabamos de crear, es decir, "**create_taggables_table.php**".

```
public function up(): void
{
    Schema::create('taggables', function (Blueprint $table) {
        $table->id();

        $table->bigInteger('tag_id')->unsigned();

        $table->morphs('taggable');
        $table->timestamps();

        $table->foreign('tag_id')->references('id')->on('tags')
            ->onDelete('cascade')
            ->onUpdate('cascade');
    });
}
```

Ilustración 45 establecer relación muchos a muchos en la migración taggables



1. La línea `$table->id();` genera de manera automática una columna llamada "id" de tipo entero sin signo, con atributos de autoincremento y como primary key.
2. La línea `$table->bigInteger('tag_id')->unsigned();` crea una columna llamada "tag_id" de tipo bigInteger (entero grande) sin signo. Esta columna se emplea para establecer una relación con otra tabla. El modificador "unsigned" garantiza que solo se almacenen valores positivos en esta columna.
3. La línea `$table->morphs('taggable');` genera automáticamente dos columnas: "taggable_id" y "taggable_type". Estas columnas se utilizan para establecer una relación polimórfica, lo que significa que el modelo asociado puede ser de diferentes tipos. Esto se logra mediante un atajo conveniente en Laravel que crea las dos columnas necesarias y configura las claves y restricciones adecuadas.
4. La línea `$table->timestamps();` crea dos columnas adicionales: "created_at" y "updated_at". Estas columnas se utilizan para registrar automáticamente la fecha y hora de creación y actualización de un registro.
5. Finalmente, `$table->foreign('tag_id')->references('id')->on('tags')->onDelete('cascade')->onUpdate('cascade');` establece una clave externa en la columna "tag_id" que hace referencia a la columna "id" en la tabla "tags". La cláusula `onDelete('cascade')` indica que si se elimina una fila en la tabla "tags", también se eliminarán las filas asociadas en esta tabla. De manera similar, `onUpdate('cascade')` indica que si se actualiza el valor de la columna "id" en la tabla "tags", también se actualizarán los valores correspondientes en esta tabla.

Asegúrate de definir las columnas y las relaciones en el archivo de migración generado de acuerdo con tus requisitos específicos para la tabla "taggables". Luego, puedes ejecutar la migración utilizando el comando `php artisan migrate` para crear la tabla en la base de datos.



2. Configuración de entidades

Introducción

Las tablas en las bases de datos a menudo tienen relaciones entre sí. Por ejemplo, una publicación de un blog puede tener muchos **comentarios**, o un pedido puede estar asociado al usuario que lo realizó. **Eloquent**, el ORM de Laravel, simplifica la gestión y el trabajo con estas relaciones y admite una variedad de relaciones comunes.

Definición de relaciones

En **Eloquent**, las relaciones se definen como métodos en las clases de tus modelos **Eloquent**. Dado que estas relaciones también actúan como generadores de **consultas** poderosos, definirlas como métodos proporciona capacidades robustas de consulta y **encadenamiento de métodos**.

A continuación, vamos a configurar las entidades en Laravel. Primero, debes ubicarte en tu entorno de desarrollo y acceder a la carpeta "App\Models". A partir de aquí, comenzaremos configurando el modelo "**Category**".

```
class Category extends Model
{
    use HasFactory;

    public function posts(): HasMany
    {
        return $this->hasMany(Post::class);
    }
}
```



- La clase "Category" extiende la clase base "Model" de Laravel, lo que implica que hereda todas las funcionalidades y características proporcionadas por el modelo base de Laravel.
- La inclusión del trait "HasFactory" señala que la clase "Category" utilizará el patrón Factory para la generación de instancias.
- La función "posts()" establece la relación "uno a muchos" entre la categoría y las publicaciones. En este contexto, se emplea el método "hasMany()" para indicar que una categoría puede tener varias publicaciones asociadas. El método "hasMany()" espera recibir como argumento el modelo relacionado, que en este caso es "Post::class".
- El tipo de retorno, HasMany, indica que la función "posts()" devuelve una instancia de la relación "uno a muchos" del tipo "HasMany". Esto permite aprovechar las funciones y métodos disponibles para la manipulación de esta relación.

Seguido de esto pasaremos a crear la relación con **Videos** el cual quedara de esta manera:

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

Ilustración 47 relación con videos

Seguimos configurando, pero ahora con el modelo "Post" en la misma carpeta "App\Models". A partir de aquí, procedemos a establecer la relación "uno a muchos" inversa, es decir, la que permite que un "Post" pertenezca a una "Categoría". Para definir la relación inversa de un "hasMany", debemos crear un método de relación en el modelo secundario que invoque al método "belongsTo".



```
class Post extends Model
{
    use HasFactory;

    public function user():BelongsTo
    {
        //Un post pertenece a un usuario
        return $this->belongsTo(User::class);
    }

    public function category():BelongsTo
    {
        //un post pertenece a una categoría
        return $this->belongsTo(Category::class);
    }
}
```

Ilustración 48 configuración en post

Ahora pasaremos a configurar las **relaciones polimórficas** del modelo Post, en el mismo archivo de configuración de la siguiente manera:

```
public function comments():MorphMany
{
    //Un post tiene muchos comentarios
    return $this->morphMany(Comment::class, 'commentable');
}

public function likes():MorphMany
{
    //Un post tiene muchos likes
    return $this->morphMany(Like::class, 'likeable');
}

public function image():MorphOne
{
    //Un post tiene una imagen
    return $this->morphOne(Image::class, 'imageable');
}

public function tags():MorphToMany
{
    //Un post tiene muchas etiquetas
    return $this->morphToMany(Tag::class, 'taggable');
}
```

Ilustración 49 configuración en post



Y así tendría que quedar el modelo 'Post':

```
8 class Post extends Model
9 {
10     use HasFactory;
11
12     public function user():BelongsTo
13     {
14         //Un post pertenece a un usuario
15         return $this->belongsTo(User::class);
16     }
17
18     public function category():BelongsTo
19     {
20         //un post pertenece a una categoría
21         return $this->belongsTo(Category::class);
22     }
23
24     public function comments():MorphMany
25     {
26         //Un post tiene muchos comentarios
27         return $this->morphMany(Comment::class, 'commentable');
28     }
29
30     public function likes():MorphMany
31     {
32         //Un post tiene muchos likes
33         return $this->morphMany(Like::class, 'likeable');
34     }
35
36     public function image():MorphOne
37     {
38         //Un post tiene una imagen
39         return $this->morphOne(Image::class, 'imageable');
40     }
41
42     public function tags():MorphToMany
43     {
44         //Un post tiene muchas etiquetas
45         return $this->morphToMany(Tag::class, 'taggable');
46     }
47 }
```



Y prácticamente el **'Video'** tiene las mismas relaciones, por lo que copiaremos y pegaremos las relaciones de post y lo pegaremos en el modelo **'Video'**, como se muestra a continuación:

```
Upp_App > app > Models > Video.php
8  class Video extends Model
9  {
10     use HasFactory;
11
12     public function user():BelongsTo
13     {
14         //Un post pertenece a un usuario
15         return $this->belongsTo(User::class);
16     }
17
18     public function category():BelongsTo
19     {
20         //un post pertenece a una categoría
21         return $this->belongsTo(Category::class);
22     }
23
24     public function comments():MorphMany
25     {
26         //Un post tiene muchos comentarios
27         return $this->morphMany(Comment::class, 'commentable');
28     }
29
30     public function likes():MorphMany
31     {
32         //Un post tiene muchos likes
33         return $this->morphMany(Like::class, 'likeable');
34     }
35
36     public function image():MorphOne
37     {
38         //Un post tiene una imagen
39         return $this->morphOne(Image::class, 'imageable');
40     }
41
42     public function tags():MorphToMany
43     {
44         //Un post tiene muchas etiquetas
45         return $this->morphToMany(Tag::class, 'taggable');
46     }
47 }
```




Cabe resaltar que en la relación polimórfica en la **migración** declaramos las entidades los cuales recuperan la entidad a relacionar.

Ahora iremos a configurar el modelo **Tag**:

```
class Tag extends Model
{
    use HasFactory;

    public function posts():MorphToMany
    {
        //Una etiqueta tiene muchos post, esta es la entidad madre
        return $this->morphedByMany(Post::class, 'taggable');
        //morphedByMany significa "transformando a muchos"
    }

    public function videos():MorphToMany
    {
        //Un video tiene muchas etiquetas
        return $this->morphedByMany(Video::class, 'taggable');
    }
}
```

Ilustración 52 configuración del modelo tag

El método "videos()" establece la relación "muchos a muchos" polimórfica entre la etiqueta y los videos. Utiliza el método "**morphedByMany()**" para indicar que una etiqueta puede estar asociada a varios videos. El modelo relacionado se especifica mediante "Video::class", y la relación se establece a través de la tabla intermedia "**taggables**".



Ahora nos dirigimos a configurar el modelo **Image**:

```
class Image extends Model
{
    use HasFactory;

    public function imageable()
    {
        //Una imagen pertenece a una entidad polimórfica
        return $this->morphTo();
        //morphTo significa "transformado a"
    }
}
```

Ilustración 53 configuración en modelo imagen

La función "imageable()" define la relación polimórfica inversa "morphTo". Esta relación implica que una imagen puede estar vinculada a diversas entidades de forma polimórfica. Al emplear el método "morphTo()", Laravel identificará de manera automática el modelo y la columna polimórfica asociada.



Ahora pasemos a configurar el modelo **Comment**:

```
class Comment extends Model
{
    use HasFactory;

    public function commentable():MorphTo
    {
        //Un comentario se transforma en una entidad polimórfica
        return $this->morphTo();
    }

    public function user():BelongsTo
    {
        //Un comentario pertenece a un usuario
        return $this->belongsTo(User::class);
    }
}
```

Ilustración 54 configuración del modelo comment

Ahora, procederemos a configurar la relación en el modelo **"Niveles"**.

La relación "muchos a través" en Laravel posibilita el acceso a los registros de una tabla relacionada a través de una tabla intermedia. En este contexto, la relación "Nivel tiene Posts a través de un usuario" implica que es posible acceder a los posts relacionados con un nivel mediante la tabla intermedia "usuario".



En el código que proporcionaste, la función "posts()" en la clase "**Level**" establece esta relación. Veamos más a detalle:

```
class Level extends Model
{
    use HasFactory;

    public function users()
    {
        //Un nivel tiene muchos usuarios
        return $this->hasMany(User::class);
    }

    public function posts()
    {
        //Relación muchos a través de usuarios con la entidad "Post"
        return $this->hasManyThrough(Post::class, User::class);
    }

    public function videos()
    {
        //Relación muchos a través de usuarios con la entidad "Video"
        return $this->hasManyThrough(Video::class, User::class);
    }
}
```

Ilustración 55 configuración en levels

La función "**posts()**" establece una relación de muchos a muchos a través de usuarios con el modelo "Post". Esto implica que un nivel puede tener múltiples posts a través de la relación con usuarios. Se utiliza el método **hasManyThrough()** para definir esta relación y se especifica el modelo relacionado ("Post::class") junto con el modelo intermedio ("User::class").

La función "**videos()**" configura una relación de muchos a muchos a través de usuarios con el modelo "Video". Esto significa que un nivel puede tener múltiples videos a través de la relación con usuarios. Se utiliza el método **hasManyThrough()** para establecer esta relación y se especifica el modelo relacionado ("Video::class") y el modelo intermedio ("User::class").



Y por último, vamos a configurar el modelo de **User**:

```
Country.php 75 public function tags()
Group.php   76 {
Image.php   77     return $this->morphToMany(Tags::class, 'taggable');
Level.php   78 }
Location.php 79
Phone.php   80 public function post():HasMany
Post.php    81 {
Profile.php 82     //Un usuario tiene muchos posts
Tag.php     83     return $this->hasMany(Post::class);
User.php    84 }
Video.php   85 public function videos():HasMany
Providers    86 {
View        87     //Un usuario tiene muchos videos
bootstrap   88     return $this->hasMany(Video::class);
config      89 }
database    90
factories   91
CategoryFactory.php 92
CommentFactory.php  93
CountryFactory.php  94
GroupFactory.php    95
ImageFactory.php    96
LevelFactory.php    97
LocationFactory.php 98
PhoneFactory.php    99
PostFactory.php     100
ProfileFactory.php  101
TagFactory.php      102
UserFactory.php     103
VideoFactory.php    104
ESQUEMA           105
                  106 public function likes():HasMany
                  107 {
                  108     //Un usuario tienen muchos likes
                  109     return $this->hasMany(Like::class, 'likeable');
                  }
                  110 public function image():MorphOne
                  111 {
                  112     //Un usuario tiene una imagen
                  113     return $this->morphOne(Image::class, 'imageable');
                  }
                  114 }
```

Ilustración 56 configuración en modelo users



10. Relación uno a muchos: Modelo User, Country, State y City

1. Creación del modelo City

Iniciamos una terminal o consola de comandos en nuestro entorno de desarrollo Laravel y ejecutamos el siguiente comando para crear el modelo:

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\xampp\htdocs\Upp_App> php artisan make:model City -mf
```

Ilustración 57 comando crear modelo city

Cuando ingreses este comando en la terminal o consola de comandos de Laravel, se creará automáticamente un modelo llamado "City" junto con una migración y una fábrica asociadas.

Luego, procederemos a definir la relación entre el modelo User y City en el archivo User.php dentro de la carpeta app\Models. Esta relación especifica que un City puede tener muchos usuarios asociados, y esto se logra utilizando el método belongsTo() de Laravel.

A continuación, encontrarás el código necesario para establecer esta relación en el modelo User en el archivo app\Models\User.php:

```
109
110     public function city()
111     {
112         return $this->belongsTo(City::class);
113     }
```

Ilustración 58 relación city y user



En la migración añadimos el campo **name** y nos referimos a el nombre de la ciudad como se muestra en esta imagen:

```
public function up(): void
{
    Schema::create('cities', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->timestamps();
    });
}
```

Ilustración 59 campo name en migración cities

Abrimos una terminal o consola de comandos en nuestro entorno de desarrollo de Laravel y procedemos a ejecutar el comando siguiente con el fin de generar un modelo:

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACION  TERMINAL  PUERTOS

PS C:\xampp\htdocs\Upp_App> php artisan make:migration add_city_id_at_users

INFO Migration [C:\xampp\htdocs\Upp_App\database\Migrations\2023_09_30_232317_add_city_id_at_users.php] created
```

Ilustración 60 comando crear modelo

Cuando ejecutamos este comando, Laravel generará automáticamente un archivo de migración con un nombre único en el directorio "database/migrations". El nombre del archivo se creará usando una marca de tiempo para asegurar su singularidad y se generará siguiendo un formato similar a "20230614091234_add_level_id_at_users.php" (donde "20230614091234" refleja la marca de tiempo actual).

Una vez que el comando se haya completado, recibirás un mensaje en la consola que confirmará la exitosa creación del archivo de migración.



Agregamos la siguiente relación en la nueva migración creada:

```
2023_09_28_022019_create_gro... 12 public function up(): void
2023_09_28_031724_create_loca... 13 {
2023_09_28_050619_create_cate... 14 Schema::table('users', function (Blueprint $table) {
2023_09_28_051600_create_post... 15     $table->bigInteger('city_id')->unsigned()->nullable()->after('id');
2023_09_28_053811_create_vide... 16
2023_09_28_055433_create_com... 17     $table->foreign('city_id')->references('id')->on('cities')->onDelete('set null')
2023_09_28_200531_create_ima... 18     ->onUpdate('cascade');
2023_09_29_020831_create_tags... 19 };
2023_09_29_023324_create_tag... 20 }
2023_09_29_033156_create_cou... 21
2023_09_30_230722_create_citie... 22 /**
2023_09_30_232317_add_city_id... 23  * Reverse the migrations.
2023_09_30_232317_add_city_id... 24  */
2023_09_30_232317_add_city_id... 25 public function down(): void
2023_09_30_232317_add_city_id... 26 {
```

Ilustración 61 relación en city_id_at_users

La instrucción `Schema::table('users', function (Blueprint $table) {` indica que se efectuarán modificaciones en la tabla 'users'. Por otro lado, `$table->bigInteger('level_id')->unsigned()->nullable()->after('id')` añade una columna adicional denominada `city_id` en la tabla 'users'. El tipo de datos seleccionado es 'bigInteger' para almacenar el valor.

`$table->bigInteger('level_id')->unsigned()->nullable()->after('id')` agrega una nueva columna llamada 'city_id' a la tabla 'users'. El tipo de datos es 'bigInteger' para almacenar números enteros grandes. La función 'unsigned()' asegura que solo se permitan valores positivos en esta columna. La opción 'nullable()' permite que la columna pueda contener valores nulos, lo que significa que no es obligatorio tener un valor de nivel asociado para cada usuario. Además, 'after('id')' especifica que la columna se posicionará después de la columna existente llamada 'id'.



`$table->foreign('city_id')->references('id')->on('cities')->onDelete('set null')->onUpdate('cascade')` establece una relación de clave externa en la columna 'city_id' de la tabla 'users'. Esto significa que la columna 'city_id' hace referencia a la columna 'id' en la tabla 'city'. La opción 'onDelete('set null')' indica que si se elimina una fila en la tabla 'cities', se establecerá automáticamente un valor nulo en la columna 'city_id' de la tabla 'users'. La opción 'onUpdate('cascade')' indica que si se realiza una actualización en la tabla 'cities', se actualizarán automáticamente todas las referencias en la tabla 'users'.

En resumen, este código de migración agrega la columna 'city_id' a la tabla 'users' y establece una relación de clave externa entre la columna 'city_id' y la tabla 'cities'. Esto permite asociar un nivel a cada usuario en la aplicación y garantizar la integridad referencial entre las tablas.

2. Creación del modelo State

Iniciamos una terminal o consola de comandos en nuestro entorno de desarrollo de Laravel y ejecutamos el siguiente comando para crear el modelo:

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\xampp\htdocs\Upp_App> php artisan make:model State -mf
```

Ilustración 62 comando para crear modelo

Cuando ingreses este comando en la terminal o consola de comandos de Laravel, se creará automáticamente un modelo llamado "State" junto con una migración y una fábrica asociadas.

Luego, procederemos a definir la relación entre el modelo City y State en el archivo City.php dentro de la carpeta app\Models. Esta relación especifica que un City puede tener muchos states asociados, y esto se logra utilizando el método belongsTo() de Laravel.



A continuación, encontrarás el código necesario para establecer esta relación en el modelo City en el archivo app\Models\City.php:

```
12     public function state()
13     {
14         return $this->belongsTo(State::class);
15     }
```

Ilustración 63 relación state y city

En la migración añadimos el campo **name** y nos referimos a el nombre del estado como se muestra en esta imagen:

```
Schema::create('states', function (Blueprint $table) {
    $table->id();
    $table->string('name');
    $table->timestamps();
});
```

Ilustración 64 campo name en migración states

Abrimos una terminal o consola de comandos en nuestro entorno de desarrollo de Laravel y procedemos a ejecutar el comando siguiente con el fin de generar un modelo:

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\xampp\htdocs\Upp_App> php artisan make:migration add_state_id_at_cities
```

Ilustración 65 comando crear modelo



Cuando ejecutamos este comando, Laravel generará automáticamente un archivo de migración con un nombre único en el directorio "database/migrations". El nombre del archivo se creará usando una marca de tiempo para asegurar su singularidad y se generará siguiendo un formato similar a "20230614091234_add_level_id_at_cities.php" (donde "20230614091234" refleja la marca de tiempo actual).

Una vez que el comando se haya completado, recibirás un mensaje en la consola que confirmará la exitosa creación del archivo de migración.

Agregamos la siguiente relación en la nueva migración creada:

```
public function up(): void
{
    Schema::table('cities', function (Blueprint $table) {
        $table->bigInteger('state_id')->unsigned()->nullable()->after('id');

        $table->foreign('state_id')->references('id')->on('states')->onDelete('set null')
        ->onUpdate('cascade');
    });
}
```

Ilustración 66 relación state_id_cities NOTA: REVISAR SINTAXIS

La instrucción `Schema::table('cities', function (Blueprint $table) {$table->bigInteger('state_id')->unsigned()->nullable()->after('id');$table->foreign('state_id')->references('id')->on('states')->onDelete('set null')->onUpdate('cascade');` añade una columna adicional denominada 'state_id' en la tabla 'cities'. El tipo de datos seleccionado es 'bigInteger' para almacenar el valor.

`$table->bigInteger('level_id')->unsigned()->nullable()->after('id')` agrega una nueva columna llamada 'state_id' a la tabla 'cities'. El tipo de datos es 'bigInteger' para almacenar números enteros grandes. La función 'unsigned()' asegura que solo se permitan valores positivos en esta columna. La opción 'nullable()' permite que la columna pueda contener valores nulos, lo que significa que no es obligatorio tener un valor de nivel asociado para cada usuario. Además, 'after('id')' especifica que la columna se posicionará después de la columna existente llamada 'id'.

`$table->foreign('state_id')->references('id')->on('states')->onDelete('set null')->onUpdate('cascade')` establece una relación de clave externa en la columna 'state_id' de la tabla 'cities'. Esto significa que la columna 'state_id' hace referencia a la columna 'id' en la tabla 'state'. La opción 'onDelete('set null')' indica que si se elimina una fila en la tabla 'states', se establecerá automáticamente un valor nulo en la columna 'states_id' de la tabla 'cities'. La



opción 'onUpdate('cascade')' indica que, si se realiza una actualización en la tabla 'states', se actualizarán automáticamente todas las referencias en la tabla 'cities'.

En resumen, este código de migración agrega la columna 'state_id' a la tabla 'cities' y establece una relación de clave externa entre la columna 'state_id' y la tabla 'states'. Esto permite asociar un nivel a cada usuario en la aplicación y garantizar la integridad referencial entre las tablas.

3. Creación del modelo Country

Iniciamos una terminal o consola de comandos en nuestro entorno de desarrollo de Laravel y ejecutamos el siguiente comando para crear los modelos:

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\xampp\htdocs\Upp_App> php artisan make:model Country -mf
```

Ilustración 67 comando para creación de modelo

Cuando ingreses este comando en la terminal o consola de comandos de Laravel, se creará automáticamente un modelo llamado "Country" junto con una migración y una fábrica asociadas.

Luego, procederemos a definir la relación entre el modelo Country y State en el archivo State.php dentro de la carpeta app\Models. Esta relación especifica que un Country puede tener muchos states asociados, y esto se logra utilizando el método belongsTo() de Laravel.

A continuación, encontrarás el código necesario para establecer esta relación en el modelo State en el archivo app\Models\State.php:

```
12     public function country()
13     {
14         return $this->belongsTo(Country::class);
15     }
```

Ilustración 68 relación en modelo state



En la migración añadimos el campo **name** y nos referimos a el nombre del país como se muestra en esta imagen:

```
public function up(): void
{
    Schema::create('countries', function (Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->timestamps();
    });
}
```

Ilustración 69 campo name en la migración contry

Abrimos una terminal o consola de comandos en nuestro entorno de desarrollo de Laravel y procedemos a ejecutar el comando siguiente con el fin de generar un modelo:

```
PS C:\xampp\htdocs\Upp_App> php artisan make:migration add_country_id_at_states

INFO Migration [C:\xampp\htdocs\Upp_App\database\Migrations\2023_10_01_022114_add_country_id_at_states.php]
```

Cuando ejecutamos este comando, Laravel generará automáticamente un archivo de migración con un nombre único en el directorio "database/migrations". El nombre del archivo se creará usando una marca de tiempo para asegurar su singularidad y se generará siguiendo un formato similar a "20230614091234_add_country_id_at_states.php" (donde "20230614091234" refleja la marca de tiempo actual).

Una vez que el comando se haya completado, recibirás un mensaje en la consola que confirmará la exitosa creación del archivo de migración.

Agregamos la siguiente relación en la nueva migración creada:

```
12 public function up(): void
13 {
14     //
15     Schema::table('states', function (Blueprint $table) {
16         $table->bigInteger('country_id')->unsigned()->nullable()->after('id');
17
18         $table->foreign('country_id')->references('id')->on('country')->onDelete('set null')
19         ->onUpdate('cascade');
```

Ilustración 70 relacion a migración add_country_id_at_states NOTA: REVISAR SINTAXIS



La instrucción `Schema::table('states', function (Blueprint $table) {$table->bigInteger('country_id')->unsigned()->nullable()->after('id');$table->foreign('country_id')->references('id')->on('country')->onDelete('set null')->onUpdate('cascade');` añade una columna adicional denominada 'country_id' en la tabla 'states'. El tipo de datos seleccionado es 'bigInteger' para almacenar el valor.

`$table->bigInteger('country_id')->unsigned()->nullable()->after('id')` agrega una nueva columna llamada 'country_id' a la tabla 'states'. El tipo de datos es 'bigInteger' para almacenar números enteros grandes. La función 'unsigned()' asegura que solo se permitan valores positivos en esta columna. La opción 'nullable()' permite que la columna pueda contener valores nulos, lo que significa que no es obligatorio tener un valor de nivel asociado para cada usuario. Además, 'after('id')' especifica que la columna se posicionará después de la columna existente llamada 'id'.

`$table->foreign('country_id')->references('id')->on('country')->onDelete('set null')->onUpdate('cascade')` establece una relación de clave externa en la columna 'country_id' de la tabla 'states'. Esto significa que la columna 'country_id' hace referencia a la columna 'id' en la tabla 'country'. La opción 'onDelete('set null')' indica que si se elimina una fila en la tabla 'countries', se establecerá automáticamente un valor nulo en la columna 'countries_id' de la tabla 'states'. La opción 'onUpdate('cascade')' indica que, si se realiza una actualización en la tabla 'country', se actualizarán automáticamente todas las referencias en la tabla 'states'.

En resumen, este código de migración agrega la columna 'country_id' a la tabla 'states' y establece una relación de clave externa entre la columna 'country_id' y la tabla 'country'. Esto permite asociar un nivel a cada usuario en la aplicación y garantizar la integridad referencial entre las tablas.



11. Relación uno a uno: Modelo User y Github_accounts

Iniciamos una terminal o consola de comandos en nuestro entorno de desarrollo de Laravel y ejecutamos el siguiente comando para crear un modelo:

```
PS C:\xampp\htdocs\Upp_App> php artisan make:model Github_accounts -mf
```

Ilustración 71 comando crear modelo

El comando "php artisan make:model X -mf" realiza las siguientes tareas:

"make:model X": Crea un nuevo modelo llamado "X". Laravel generará un archivo denominado "X.php" dentro de la carpeta "app" de tu proyecto. Este modelo se emplea para la interacción con los datos en la tabla correspondiente de la base de datos.

"-m": Genera una migración para el modelo "X". Laravel generará automáticamente un archivo de migración en la carpeta "database/migrations". La migración contendrá métodos para crear la tabla de la base de datos que corresponde al modelo "X". Posteriormente, puedes personalizar la migración según tus necesidades para agregar o modificar columnas.

"-f": Genera una factory para el modelo "X". Laravel creará un archivo de factory en la carpeta "database/factories". Las factories se emplean para generar datos de prueba para el modelo, lo que te permitirá crear registros ficticios de manera rápida y sencilla.

En resumen, el comando "php artisan make:model X -mf" te facilita la creación de un modelo "X", una migración para la tabla asociada y una factory para generar datos de prueba



relacionados con dicho modelo. Esto te proporciona una estructura básica para comenzar a trabajar con el modelo "X" en tu proyecto de Laravel.

"A continuación, vamos a establecer la relación entre el modelo 'User' y 'Github_accounts' dentro del archivo del modelo 'User'. Esta relación determina que un usuario está vinculado a un perfil utilizando la función 'hasOne()' proporcionada por Laravel."

Luego, se procede a dirigirse al archivo 'app\Models\User.php' y se escribe el siguiente código.

```
114
115     public function github_accounts()
116     {
117         return $this->hasOne(Github_accounts::class);
118     }
```

Ilustración 72 relación en modelo user

Este código define la relación entre el modelo User y el modelo Github_accounts en Laravel. El método hasOne() establece que un usuario tiene una cuenta asociada.

Ahora iremos a configurar los campos para la migración en la tabla "Github_accounts" el cual se encuentran en "database\Migrations\profiles_table.php"

```
public function up(): void
{
    Schema::create('github_accounts', function (Blueprint $table) {
        $table->id();

        $table->bigInteger('user_id')->unsigned(); #Codigo añadido

        $table->string('github_id')->nullable();
        $table->string('github_token')->nullable();
        $table->string('github_refresh_token')->nullable();

        $table->timestamps();

        $table->foreign('user_id')->references('id')->on('users')
            ->onDelete('cascade')
            ->onUpdate('cascade');
    });
}
```

Ilustración 73 campos en la migración hithub_accounts_table



"\$table->bigInteger('user_id')->unsigned();" Este comando añade una columna denominada "user_id" de tipo bigInteger (entero grande) y sin signo a la tabla. Esta columna se utiliza para establecer la relación con el modelo User.

Luego, se añade una restricción de clave externa a la columna "user_id" utilizando el siguiente código:

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

Esta restricción de clave externa conecta la columna "user_id" con la columna "id" en la tabla "users". Esto establece una relación de clave externa entre las tablas "github_accounts" y "users", de modo que cualquier eliminación o actualización de un usuario en la tabla "users" se reflejará automáticamente en la tabla "github_accounts".

Ejecutamos el comando de las migraciones:

```
PS C:\xampp\htdocs\Upp_App> php artisan migrate
```

```
INFO Running migrations.
```

```
2023_09_28_050619_create_categories_table ..... 88ms DONE  
2023_09_28_051600_create_posts_table ..... 141ms DONE  
2023_09_28_053811_create_videos_table ..... 109ms DONE  
2023_09_28_055433_create_comments_table ..... 102ms DONE  
2023_09_28_200531_create_images_table ..... 23ms DONE  
2023_09_29_020831_create_tags_table ..... 10ms DONE  
2023_09_29_023324_create_taggables_table ..... 76ms DONE
```




Y este sería el resultado final de todas las relaciones a nivel de base de datos:

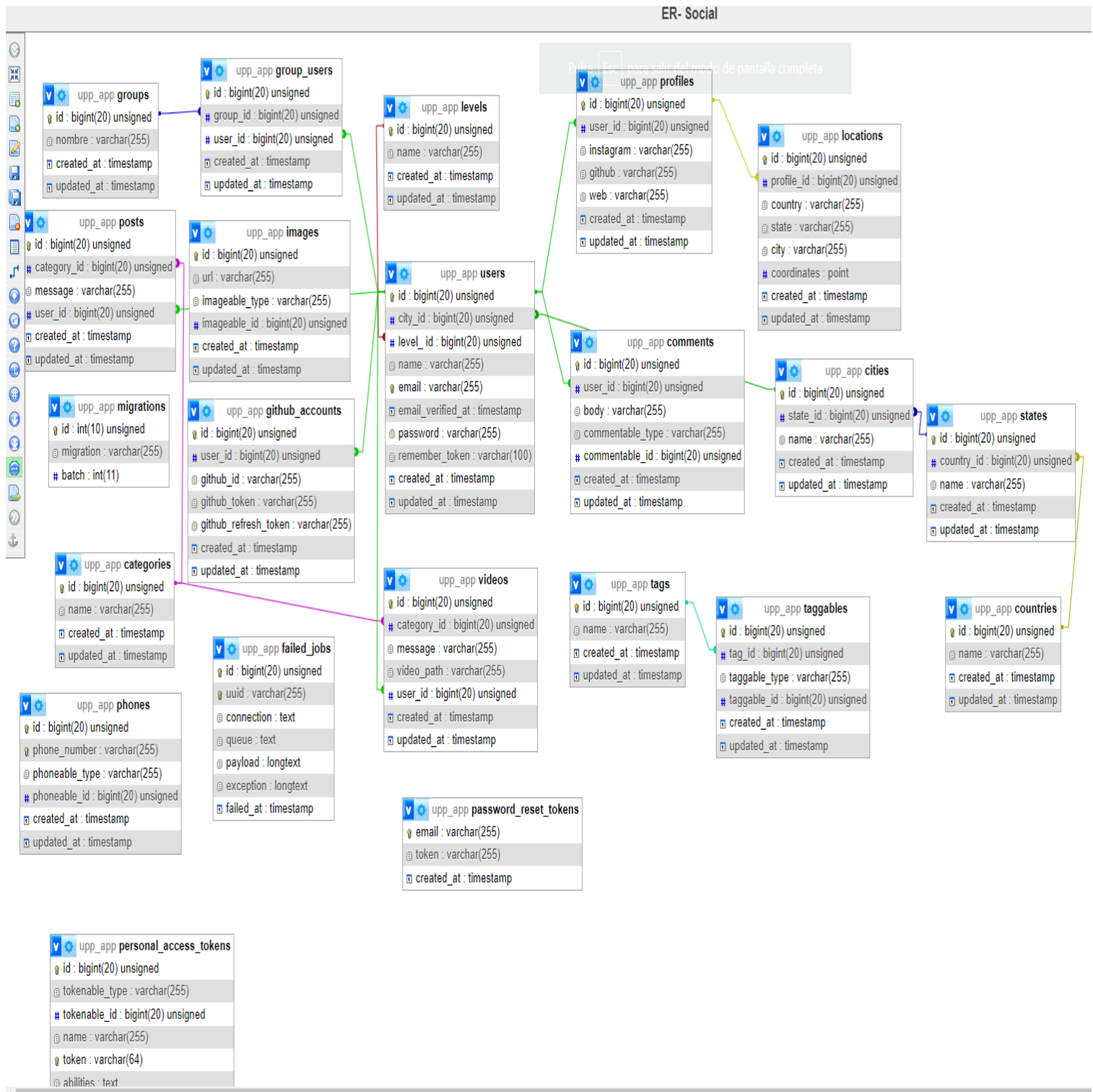


Ilustración 74 diagrama er final