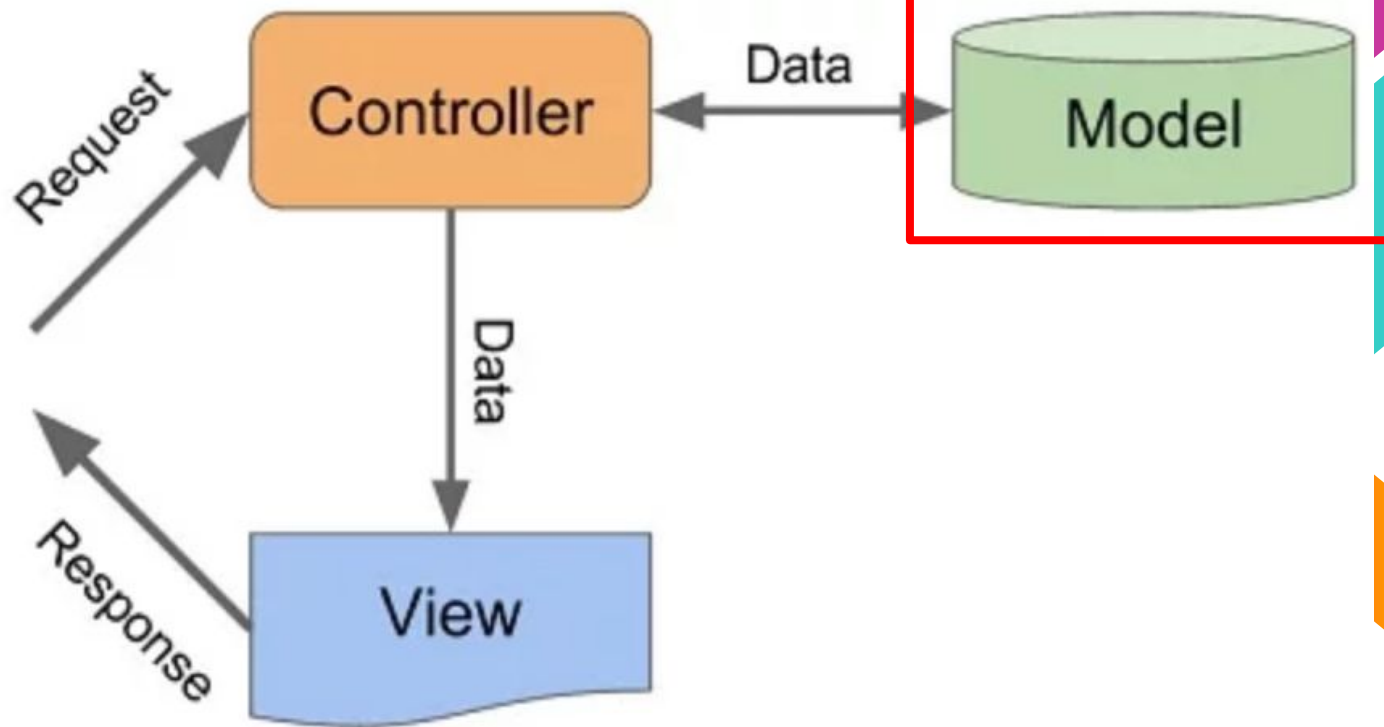


Abstract geometric shapes in the top-left corner, including a light blue parallelogram, a green parallelogram, a brown parallelogram, and a large purple parallelogram.

LARAVEL

CLASE 04

Abstract geometric shapes in the top-right corner, including a light blue parallelogram, a green parallelogram, a purple parallelogram, and a red parallelogram.





MODEL

El modelo representa la lógica por debajo de nuestra aplicación que muchas veces se condice con nuestra capa de datos. Dicho de otra manera, suelen ser clases que se condicen con nuestras tablas en la base de datos.



¿Cómo creamos un modelo?

Desde la consola de comandos, ejecutamos el siguiente código:

```
php artisan make:model NombreModelo
```



```
// app/Pelicula.php
```

```
namespace App;
```

```
use Illuminate\Database\Eloquent\Model;
```

```
class Pelicula extends Model {
```

```
    /**
```

```
    * The attributes that aren't mass assignable
```

```
    *
```

```
    * @var array
```

```
    */
```

```
    protected $guarded = [];
```

```
    /**
```

```
    * The attributes that should be mutated to dates.
```

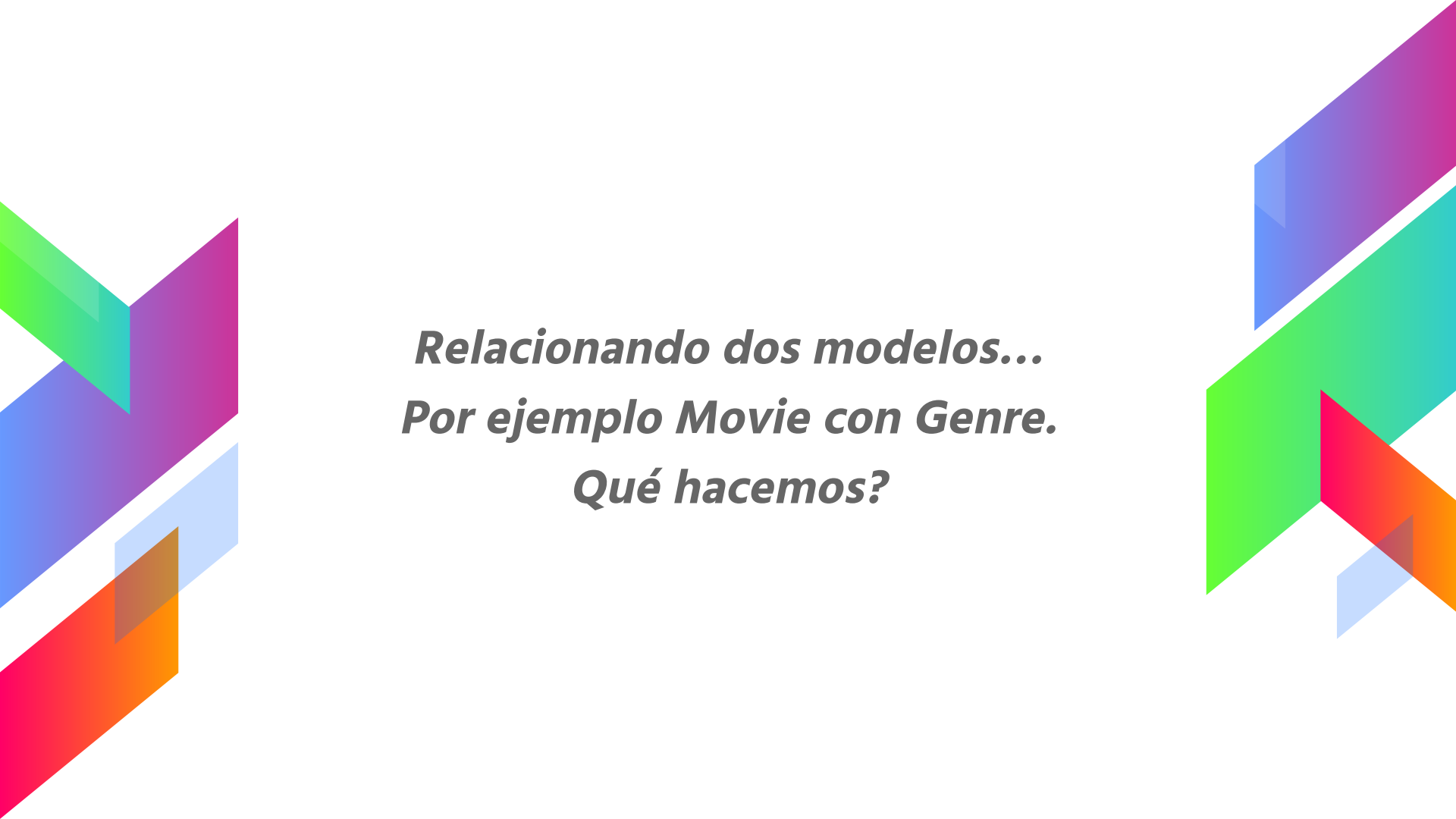
```
    *
```

```
    * @var array
```

```
    */
```

```
    protected $dates = ['fecha_de_estreno'];
```

```
}
```

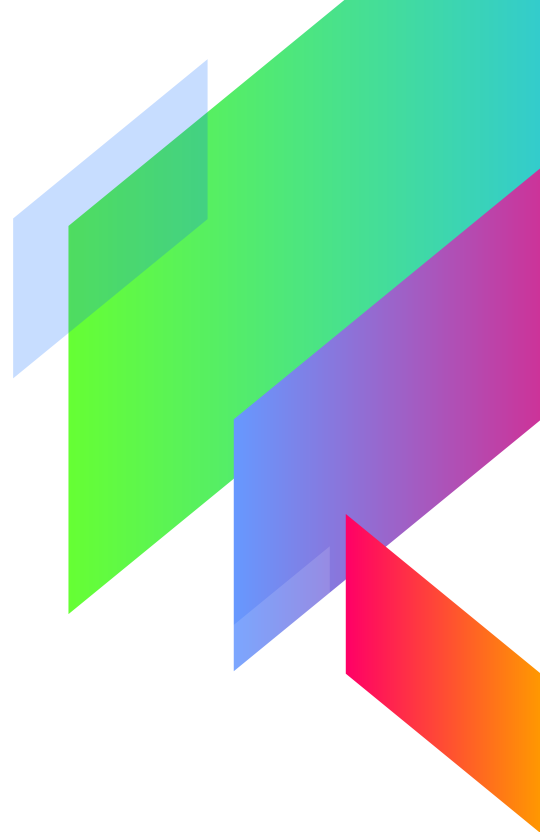


***Relacionando dos modelos...
Por ejemplo Movie con Genre.
Qué hacemos?***



RELACIONES

Manejo de relaciones en Laravel



RELACIONES

DIRECCIONALIDAD

A -- conoce a --> B

No es lo mismo que...

B -- conoce a --> A

CARDINALIDAD

A -- tiene un --> B

No es lo mismo que...

A -- tiene muchos --> B

ELOQUENT: RELACIONES

N:1 - Belongs To

- ✗ Episode **belongsTo** Season
- ✗ Profile **belongsTo** User

1:0...1 - Has One

- ✗ User **hasOne** Profile
- ✗ Person **hasOne** DNI

1:0...N - Has Many

- ✗ Season **hasMany** Episode
- ✗ Genre **hasMany** Movie

N:M - Belongs to Many

- ✗ Actor **belongsToMany** Movie
- ✗ Client **belongsToMany** Product

¿Cómo creamos una relación?

Por ejemplo tenemos dos clases: **User** y **Profile**

User	Profile
- id	- id
- email	- user_id
- password	- name
	- address

Usando Eloquent creamos la relación donde la necesitamos, para ello creamos un método (con el nombre que queramos), pero por convención podemos ponerle el nombre de la otra clase:

```
class Profile extends Model
{
    public function user() {
        //aquí definimos la relación
    }
}
```

Definiendo una relación

User

- id
- email
- password

Profile

- id
- user_id
- name
- address

```
class Profile extends Model
{
    public function user() {
        return $this->belongsTo( User::class );
    }
}
```

Dado que seguimos las convenciones de nombres de laravel con respecto a la BD, ya con esa definición tenemos nuestra relación.
Sino tuviesemos que especificar los campos de la BD.

Suponiendo que los campos se llamen:

User

- id
- email
- password

Profile

- id
- user_id
- name
- address

```
class Profile extends Model
{
    public function user() {
        return $this->belongsTo( User::class, 'user_id', 'id' );
    }
}
```

Como segundo parámetro le especifico el campo en el modelo Profile. Hasta el momento Laravel todavía asume que la llave primaria del Modelo User es "id", pero si es distinto, hay que aclarárselo como **tercer parámetro** de la relación.

```
class User extends Model
{
    public function profile()
    {
        return $this->hasOne(Profile::class, 'id', 'user_id');
    }
}
```

```
class Profile extends Model
{
    public function welcome(){
        return 'Welcome '.$this->name;
    }
}
```

```
// A partir de un Objeto User
$user = User::find(1);
```

```
// Obtenemos su objeto Profile relacionado
$profile = $user->profile; //se usa sin los parentesis ya que es una relación de Eloquent
```

```
//entonces podemos acceder a los atributos o métodos de ese objeto Profile
$profile->name;
$profile->welcome();
```

```
class User extends Model
{
}
```

```
class Profile extends Model
{
    public function user()
    {
        return $this->belongsTo(User::class, 'user_id', 'id');
    }
}
```

```
//buscamos un perfil en la DB
$profile = Profile::find(42);
```

```
//pedimos su usuario
$user = $profile->user;
```

```
class User extends Model
{
    public function pets()
    {
        return $this->hasMany(Pet::class, 'user_id', 'id');
    }
}

class Pet extends Model
{
    public function user()
    {
        return $this->belongsTo(User::class, 'user_id', 'id');
    }
}
```

// A partir de un Objeto User obtenemos su colección de objetos Pet relacionados

```
$user = User::find(1);
$pets = $user->pets;
```

// Y viceversa

```
$pet = Pet::find(9);
$user = $pet->user;
```

// Podemos hacer una query para obtener solo algunos

```
$dogs = $user->pets()->where('type', 'dog')->get();
```

```
class Actor extends Model
{
    public function movies()
    {
        return $this->belongsToMany(Movie::class, 'table', 'foreign_key', 'other_key');
    }
}

class Movie extends Model
{
    public function actors()
    {
        return $this->belongsToMany(Actor::class, 'table', 'foreign_key', 'other_key');
    }
}
```

```
// A partir de un Objeto Actor obtenemos su colección de objetos Movies relacionados
$actor = Actor::find(1);
$movies = $actor->movies;
```

```
// Y viceversa
$movie = Movie::find(23);
$actors = $movie->actors;
```

```
// También podemos hacer queries
$bestActors = $movie->actors()->where('rating', '>', 8)->orderBy('last_name')->get();
```


**ES MOMENTO DE
PRACTICAR !**





Query Builder

Generador de consultas SQL



// Obtener los datos de una tabla

```
$user = DB::table('users')->get();
```

// Aplicar la clausula select o distinct

```
$users = DB::table('users')->select('name')->get();
```

```
$users = DB::table('users')->select('name as user_name')->get();
```

```
$users = DB::table('users')->distinct()->get();
```

// Aplicar el operador Where

```
$users = DB::table('users')->where('votes', '>', 100)->get();
```

```
$users = DB::table('users')->where('votes', '>', 100)->orWhere('name', 'John')->get();
```

```
$users = DB::table('users')->whereBetween('votes', array(1, 100))->get();
```

```
$users = DB::table('users')->whereIn('id', array(1, 2, 3))->get();
```

```
$users = DB::table('users')->whereNotIn('id', array(1, 2, 3))->get();
```

```
$users = DB::table('users')->whereNull('updated_at')->get();
```

// Aplicar order by, group by, y having

```
$users = DB::table('users')  
    ->orderBy('name', 'desc')  
    ->groupBy('count')  
    ->having('count', '>', 100)  
    ->get();
```

// Aplicar offset, y limit

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

// Aplicar join

```
$users = DB::table('users')  
    ->join('contacts', 'users.id', '=', 'contacts.user_id')  
    ->join('orders', 'users.id', '=', 'orders.user_id')  
    ->select('users.id', 'contacts.phone', 'orders.price')  
    ->get();
```

```
$users = DB::table('users')  
    ->leftJoin('posts', 'users.id', '=', 'posts.user_id')  
    ->get();
```



¡Cuidado!

Nótese que al utilizar el **Query Builder** siempre finalizamos con un método **->get()** // **->first()** // **->value()**



The image features abstract geometric shapes in the corners. On the left, there are overlapping shapes in shades of green, blue, orange, and purple. On the right, there are overlapping shapes in shades of green, blue, purple, and orange. The central text is in a bold, black, sans-serif font.

**¡ HASTA LA
PROXIMA !**