Eloquent: Relaciones

Introducción

Las tablas de base de datos frecuentemente están relacionadas a otra tabla. Por ejemplo, un post de un blog puede tener muchos comentarios o un pedido podría estar relacionado al usuario que lo ordenó. Eloquent hace que manejar y trabajar con estas relaciones sea fácil y soporta varios tipos de relaciones:

- Uno a Uno
- Uno a Muchos
- Muchos a Muchos
- Uno a Través de
- Muchos a Través de
- Uno a Uno (Polimórfica)
- Uno a Muchos (Polimórfica)
- Muchos a Muchos (Polimórfica)

Definiendo relaciones

Las relaciones de Eloquent se definen como métodos en tus clases de modelo de Eloquent. Debido a que, como los mismos modelos Eloquent, las relaciones también sirven como poderosos constructores de consultas, puesto que definir relaciones como

ejemplo, podemos encadenar restricciones adicionales en esta relación `posts`:

```php
$user->posts()->where('active', 1)->get(
```

Pero, antes de profundizar demasiado en el uso de relaciones, aprendamos cómo definir cada tipo.

Uno A Uno

Una relación de uno a uno es una relación muy sencilla. Por ejemplo, un modelo `User` podría estar asociado con un `Phone`. Para definir esta relación, colocaremos un método `phone` en el modelo `User`. El método `phone` debería llamar al método `hasOne` y devolver su resultado:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
    * Get the phone record associated w
    */
    public function phone()
    {
        return $this->hasOne('App\Phone
```

El primer argumento pasado al método `hasOne` es el nombre del modelo relacionado. Una vez que la relación es definida, podemos obtener el registro relacionado usando propiedades dinámicas de Eloquent. Las propiedades dinámicas permiten que accedas a métodos de relación como si fueran propiedades definidas en el modelo:

```php
$phone = User::find(1)->phone;
```

Eloquent determina la clave foránea de la relación en base al nombre del modelo. En este caso, se asume automáticamente que el modelo `Phone` tenga una clave foránea `user_id`. Si deseas sobrescribir esta convención, puedes pasar un segundo argumento al método `hasOne`:

```php
return $this->hasOne('App\Phone', 'fore
```

Adicionalmente, Eloquent asume que la clave foránea debería tener un valor que coincida con la columna `id` (o `$primaryKey` personalizada) del padre. En otras palabras, Eloquent buscará el valor de la columna `id` del usuario en la columna `user_id` de `Phone`. Si prefieres que la relación use un valor distinto de `id`, puedes pasar un tercer argumento al método `hasOne` especificando tu clave personalizada:

Definiendo el inverso de la relación

Así, podemos acceder al modelo `Phone` desde nuestro `User`. Ahora, vamos a definir una relación en el modelo `Phone` que nos permitirá accdeder al `User` que posee el teléfono. Podemos definir el inverso de una relación `hasOne` usando el método `belongsTo`:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model
{
    /**
    * Get the user that owns the phone.
    */
    public function user()
    {
        return $this->belongsTo('App\Use
    }
}
```

En el ejemplo anterior, Eloquent intentará hacer coincidir el `user_id` del modelo `Phone` con un `id` en el modelo `User`. Eloquent determina el nombre de la clave foránea de forma predeterminada al examinar el nombre del método de la relación y

`Phone` no es `user_id`, puedes pasar un nombre de clave personalizada como segundo argumento al método `belongsTo`:

```php
/**
* Get the user that owns the phone.
*/
public function user()
{
    return $this->belongsTo('App\User',
}
```

Si tu modelo padre no usa `id` como su clave primaria, o deseas hacer join al modelo hijo con una columna diferente, puedes pasar un tercer argumento al método `belongsTo` especificando la clave personalizada de tu tabla padre:

```php
/**
* Get the user that owns the phone.
*/
public function user()
{
    return $this->belongsTo('App\User',
}
```

Uno a muchos

Una relación de "uno-a-muchos" es usada para definir relaciones donde un solo modelo posee cualquier cantidad de otros modelos. Por ejemplo, un

de Eloquent, las relaciones uno-a-muchos son definidas al colocar una función en tu modelo Eloquent:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
    * Get the comments for the blog post
    */
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

Recuerda, Eloquent determinará automáticamente la columna de clave foránea apropiada en el modelo `Comment`. Por convención, Eloquent tomará el nombre "snake_case" del modelo que la contiene y le agregará el sufijo `_id`. Para este ejemplo, Eloquent asumirá que la clave foránea del modelo `Comment` es `post_id`.

Una vez que la relación ha sido definida, podemos acceder a la colección de comentarios al acceder a la propiedad `comments`. Recuerda, ya que Eloquent

definidos como propiedades en el modelo:

```php
$comments = App\Post::find(1)->comments;

foreach ($comments as $comment) {
    //
}
```

Debido a que todas las relaciones también sirven como constructores de consultas (query builders), puedes agregar restricciones adicionales a cuyos comentarios sean obtenidos ejecutando el método `comments` y encadenando condiciones en la consulta:

```php
$comment = App\Post::find(1)->comments()
```

Igual que el método `hasOne`, también puedes sobrescribir las claves foráneas y locales al pasar argumentos adicionales al método `hasMany`:

```php
return $this->hasMany('App\Comment', 'fo

return $this->hasMany('App\Comment', 'fo
```

Uno a muchos (inverso)

Ahora que puedes acceder a todos los comentarios de un post, vamos a definir una relación para permitir

función de relación en el modelo hijo que ejecute el método `belongsTo`:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
    * Get the post that owns the comment
    */
    public function post()
    {
        return $this->belongsTo('App\Post');
    }
}
```

Una vez que la relación ha sido definida, podemos obtener el modelo `Post` para un `Comment` accediendo a la "propiedad dinámica" de `post`:

```php
$comment = App\Comment::find(1);

echo $comment->post->title;
```

En el ejemplo anterior, Eloquent tratará de hacer coincidir el `post_id` del modelo `Comment` con un `id` en el modelo `Post`. Eloquent determina el

sufijo `_` al nombre del método, seguido del nombre de la columna principal de la llave. Sin embargo, si la clave foránea en el modelo `Comment` no es `post_id`, puedes pasar un nombre de clave personalizado como segundo argumento al método `belongsTo`:

```php
/**
* Get the post that owns the comment.
*/
public function post()
{
    return $this->belongsTo('App\Post',
}
```

Si tu modelo padre no usa `id` como su clave primaria, o deseas hacer join al modelo hijo con una columna diferente, puedes pasar un tercer argumento al método `belongsTo` especificando la clave personalizada de tu tabla padre.

```php
/**
* Get the post that owns the comment.
*/
public function post()
{
    return $this->belongsTo('App\Post',
}
```

Muchos a muchos

`hasMany`. Un ejemplo de tal relación es un usuario con muchos roles, donde los roles también son compartidos por otros usuarios. Por ejemplo, muchos usuarios pueden tener el rol "Admin". Para definir esta relación, tres tablas de bases de datos son necesitadas: `users`, `roles`, y `role_user`. La tabla `role_user` es derivada del orden alfabético de los nombres de modelo relacionados y contiene las columnas `user_id` y `role_id`.

Las relaciones de muchos-a-muchos son definidas escribiendo un método que devuelve el resultado del método `belongsToMany`. Por ejemplo, vamos a definir el método `roles` en nuestro modelo `User`:

php

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

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

`roles` :

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

foreach ($user->roles as $role) {
    //
}
```

Como con los otros tipos de relación, puedes ejecutar el método `roles` para continuar encadenando las restricciones de consulta en la relación:

```php
$roles = App\User::find(1)->roles()->or
```

Como mencionamos previamente, para determinar el nombre de la tabla asociativa, Eloquent juntará los dos nombres de modelo en orden alfabético. Sin embargo, eres libre de sobrescribir esta convención. Puedes hacer eso pasando un segundo argumento al método `belongsToMany` :

```php
return $this->belongsToMany('App\Role',
```

Además de personalizar el nombre de la tabla asociativa, también puedes personalizar los nombres de columna de las claves en la tabla pasando argumentos adicionales al método `belongsToMany` . El tercer argumento es el nombre de clave foránea del modelo en el cual estás

estás asociando:

```php
return $this->belongsToMany('App\Role',
```

Definiendo el inverso de la relación

Para definir el inverso de una relación de muchos-a-muchos, puedes colocar otra llamada de `belongsToMany` en tu modelo relacionado. Para continuar con nuestro ejemplo de roles de usuario, vamos a definir el método `users` en el modelo `Role`:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
    * The users that belong to the role
    */
    public function users()
    {
        return $this->belongsToMany('App
    }
}
```

Como puedes ver, la relación es definida

modelo `App\User`. Ya que estamos reusando el método `belongsToMany`, todas las tablas y opciones de personalización de claves usuales están disponibles al momento de definir el inverso de las relaciones de muchos-a-muchos.

Obteniendo columnas de tablas intermedias (Pivote)

Como ya has aprendido, trabajar con relaciones de muchos-a-muchos requiere la presencia de una tabla intermedia o pivote. Eloquent proporciona algunas formas muy útiles de interactuar con esta tabla. Por ejemplo, vamos a asumir que nuestro objeto `User` tiene muchos objetos `Role` al que está relacionado. Después de acceder a esta relación, podemos acceder a la tabla intermedia usando el atributo `pivot` en los modelos:

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

foreach ($user->roles as $role) {
    echo $role->pivot->created_at;
}
```

Ten en cuenta que a cada modelo `Role` que obtenemos se le asigna automáticamente un atributo `pivot`. Este atributo contiene un modelo que representa la tabla intermedia y puede ser usado como cualquier otro modelo de Eloquent.

pivote contiene atributos extras, debes especificarlos cuando definas la relación.

```php
return $this->belongsToMany('App\Role');
```

Si quieres que tu tabla pivote automáticamente mantenga las marcas de tiempo `created_at` y `updated_at`, usa el método `withTimestamps` en la definición de la relación:

```php
return $this->belongsToMany('App\Role');
```

Personalizando el nombre del atributo `pivot`

Como se señaló anteriormente, los atributos de la tabla intermedia pueden ser accedidos en modelos usando el atributo `pivot`. Sin embargo, eres libre de personalizar el nombre de este atributo para que refleje mejor su propósito dentro de tu aplicación.

Por ejemplo, si tu aplicación contiene usuarios que pueden suscribirse a podcasts, probablemente tengas una relación de muchos-a-muchos entre usuarios y podcasts. Si éste es el caso, puedes desear renombrar tu tabla pivote intermedia como `subscription` en lugar de `pivot`. Esto puede ser hecho usando el método `as` al momento de definir la relación:

```
->withTimestamps();
```

Una vez hecho esto, puedes acceder a los datos de la tabla intermedia usando el nombre personalizado:

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

foreach ($users->flatMap->podcasts as $p
    echo $podcast->subscription->create
}
```

Filtrando relaciones a través de columnas de tablas intermedias

También puedes filtrar los resultados devueltos por `belongsToMany` usando los métodos `wherePivot` y `wherePivotIn` al momento de definir la relación:

```php
return $this->belongsToMany('App\Role')

return $this->belongsToMany('App\Role')
```

Definiendo modelos de tabla intermedia personalizados

Si prefieres definir un modelo personalizado para representar la tabla intermedia o pivote de tu relación, puedes ejecutar el método `using` al momento de definir la relación. Los modelos de

`Illuminate\Database\Eloquent\Relations\Pivot` mientras que los modelos polimórficos muchos-a-muchos deben extender la clase `Illuminate\Database\Eloquent\Relations\MorphPivot`. Por ejemplo, podemos definir un `Role` que use un modelo pivote `RoleUser` personalizado:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
    * The users that belong to the role
    */
    public function users()
    {
        return $this->belongsToMany('App
    }
}
```

Al momento de definir el modelo `RoleUser`, extenderemos la clase `Pivot`:

```php
<?php

namespace App;
```

```
class RoleUser extends Pivot
{
    //
}
```

Puedes combinar `using` y `withPivot` para retornar columnas de la tabla intermedia. Por ejemplo, puedes retornar las columnas `created_by` y `updated_by` desde la tabla pivote `RoleUser` pasando los nombres de las columnas al método `withPivot`:

php

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
    * The users that belong to the role
    */
    public function users()
    {
        return $this->belongsToMany('App
                    ->using('App\Ro
                    ->withPivot([
                        'created_by
                        'updated_by
                    ]);
```

> **Nota**
>
> **Nota:** Los modelos Pivot no pueden usar el trait `SoftDeletes`. Si necesitas hacer soft delete de registros pivot considera convertir tu modelo pivot a un modelo de Eloquent.

Modelos de pivote personalizados e IDs incrementales

Si has definido una relación de muchos a muchos que usa un modelo de pivote personalizado, y ese modelo de pivote tiene una clave primaria de incremento automático, debes asegurarte de que su clase de modelo de pivote personalizado defina una propiedad `incrementing` que se establece en `true`.

php

```
/**
* Indicates if the IDs are auto-increme
*
* @var bool
*/
public $incrementing = true;
```

Tiene uno a través de (hasOneThrough)

si cada proveedor (supplier) tiene un usuario (user) y cada usuario está asociado con un registro del historial (history) de usuarios, entonces el modelo del proveedor puede acceder al historial del usuario *a través* del usuario. Veamos las tablas de base de datos necesarias para definir esta relación:

```php
users
    id - integer
    supplier_id - integer

suppliers
    id - integer

history
    id - integer
    user_id - integer
```

Aunque la tabla `history` no contiene una columna `supplier_id`, la relación `hasOneThrough` puede proporcionar acceso al historial del usuario desde el modelo del proveedor. Ahora que hemos examinado la estructura de la tabla para la relación, vamos a definirla en el modelo `Supplier`:

```php
<?php

namespace App;
```

```
class Supplier extends Model
{
    /**
    * Get the user's history.
    */
    public function userHistory()
    {
        return $this->hasOneThrough('Ap
    }
}
```

El primer argumento pasado al método `hasOneThrough` es el nombre del modelo final al que queremos acceder, mientras que el segundo argumento es el nombre del modelo intermedio.

Se utilizarán las convenciones típicas de clave foránea de Eloquent al realizar las consultas de la relación. Si deseas personalizar las claves de la relación, puedes pasarlas como el tercer y cuarto argumento al método `hasOneThrough`. El tercer argumento es el nombre de la clave foránea en el modelo intermedio. El cuarto argumento es el nombre de la clave foránea en el modelo final. El quinto argumento es la clave local, mientras que el sexto argumento es la clave local del modelo intermedio:

```php
class Supplier extends Model
{
    /**
```

```
    public function userHistory()
    {
        return $this->hasOneThrough(
            'App\History',
            'App\User',
            'supplier_id', // Foreign ke
            'user_id', // Foreign key or
            'id', // Local key on suppl
            'id' // Local key on users t
        );
    }
}
```

Tiene muchos a través de (hasManyThrough)

La relación "tiene-muchos-a-través-de" proporciona una abreviación conveniente para acceder a relaciones distantes por medio de una relación intermedia. Por ejemplo, un modelo `Country` podría tener muchos modelos `Post` a través de un modelo `User` intermedio. En este ejemplo, podrías traer todos los posts de un blog para un país dado. Vamos a buscar las tablas requeridas para definir esta relación:

php

```
countries
    id - integer
    name - string

users
```

```
    name - string

posts
    id - integer
    user_id - integer
    title - string
```

Aunque los `posts` no contienen una columna `country_id`, la relación `hasManyThrough` proporciona acceso a los posts de un país por medio de `$country->posts`. Para ejecutar esta consulta, Eloquent inspecciona el `country_id` de la tabla intermedia `users`. Después de encontrar los ID de usuario que coincidan, serán usados para consultar la tabla `posts`.

Ahora que hemos examinado la estructura de la tabla para la relación, vamos a definirla en el modelo `Country`:

php

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Country extends Model
{
    /**
    * Get all of the posts for the count
    */
    public function posts()
```

```
    }
}
```

El primer argumento pasado al método `hasManyThrough` es el nombre del modelo final que deseamos acceder, mientras que el segundo argumento es el nombre del modelo intermedio.

Las convenciones de clave foránea típicas de Eloquent serán usadas al momento de ejecutar las consultas de la relación. Si prefieres personalizar las claves de la relación, puedes pasarlos como tercer y cuarto argumentos del método `hasManyThrough`. El tercer argumento es el nombre de la clave foránea en el modelo intermedio. El cuarto argumento es el nombre de la clave foránea en el modelo final. El quinto argumento es la clave local, mientras el sexto argumento es la clave local del modelo intermedio:

```php
class Country extends Model
{
    public function posts()
    {
        return $this->hasManyThrough(
            'App\Post',
            'App\User',
            'country_id', // Foreign key
            'user_id', // Foreign key on
            'id', // Local key on countr
            'id' // Local key on users t
        );
```

Relaciones polimórficas

Una relación polimórfica permite que el modelo objetivo pertenezca a más de un tipo de modelo usando una sola asociación.

Una a una (polimórfica)

Estructura de tabla

Una relación polimorfica de uno-a-uno es similar a una relación de uno-a-uno simple; sin embargo, el modelo objetivo puede pertenecer a más de un tipo de modelo en una sola asociación. Por ejemplo, un `Post` de un blog y un `User` pueden compartir una relación polimórfica con un modelo `Image`. Usando una relación polimórfica de uno-a-uno te permite tener una sola lista de imagenes únicas que son usadas tanto los posts del blog como por las cuentas de los usuarios. Primero, vamos a examinar la estructura de la tabla:

```php
posts
    id - integer
    name - string

users
    id - integer
```

```
images
    id - integer
    url - string
    imageable_id - integer
    imageable_type - string
```

Observa las columnas `imageable_id` y `imageable_type` en la tabla `images`. La columna `imageable_id` contendrá el valor del ID del post o el usuario, mientras que la columna `imageable_type` contendrá el nombre de clase del modelo padre. La columna `imageable_type` es usada por Eloquent para determinar cuál "tipo" de modelo padre retornar al acceder a la relación `imageable`.

Estructura del modelo

A continuación, vamos a examinar las definiciones de modelo necesarias para construir esta relación:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Image extends Model
{
    /**
    * Get the owning imageable model.
```

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

class Post extends Model
{
    /**
    * Get the post's image.
    */
    public function image()
    {
        return $this->morphOne('App\Imag
    }
}

class User extends Model
{
    /**
    * Get the user's image.
    */
    public function image()
    {
        return $this->morphOne('App\Imag
    }
}
```

Retornando la relación

Una vez que tu base de datos y modelos son definidos, puedes acceder a las relaciones mediante tus modelos. Por ejemplo, para retornar la imagen para un post, podemos usar la propiedad dinámica

```php
$post = App\Post::find(1);

$image = $post->image;
```

Puedes también retornar el padre del modelo polimórfico accediendo al nombre del método que realiza la llamada a `morphTo`. En nuestro caso, éste es el método `imageable` en el modelo `Image`. Entonces, accederemos al método como una propiedad dinámica:

```php
$image = App\Image::find(1);

$imageable = $image->imageable;
```

La relación `imageable` en el modelo `Image` retornar ya sea una instancia de `Post` o `User`, dependiendo del tipo de modelo que posea la imagen.

Uno a muchos (Polimórfica)

Estructura de tabla

Una relación polimórfica de uno-a-muchos es similar a una relación de uno-a-muchos sencilla; sin embargo, el modelo objetivo puede pertenecer a más de un tipo de modelo en una sola asociación. Por ejemplo, imagina que usuarios de tu aplicación pueden comentar tanto en posts como en videos.

vamos a examinar la estructura de tabla requerida para esta relación:

```php
posts
    id - integer
    title - string
    body - text

videos
    id - integer
    title - string
    url - string

comments
    id - integer
    body - text
    commentable_id - integer
    commentable_type - string
```

Estructura de modelo

A continuación, vamos a examinar las definiciones de modelos necesarias para construir esta relación:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
```

```
     * Get the owning commentable model.
     */
    public function commentable()
    {
        return $this->morphTo();
    }
}

class Post extends Model
{
    /**
     * Get all of the post's comments.
     */
    public function comments()
    {
        return $this->morphMany('App\Cor
    }
}

class Video extends Model
{
    /**
     * Get all of the video's comments.
     */
    public function comments()
    {
        return $this->morphMany('App\Cor
    }
}
```

Retornando la relación

Una vez que tu base de datos y modelos son definidos, puedes acceder a las relaciones mediante

dinámica `comments` :

```php
$post = App\Post::find(1);

foreach ($post->comments as $comment) {
    //
}
```

También puedes retornar al propietario de una relación polimórfica desde un modelo polimórfico accediendo al nombre del método que realiza la llamada a `morphTo` . En nuestro caso, éste es el método `commentable` en el modelo `Comment` . Así que, accederemos a dicho método como una propiedad dinámica:

```php
$comment = App\Comment::find(1);

$commentable = $comment->commentable;
```

La relación `commentable` en el modelo `Comment` retornará ya sea una instancia `Post` o `Video` , dependiendo de qué tipo de modelo es el propietario del comentario.

Muchos A Muchos (Polimórfica)

Estructura de tabla

Las relaciones polimórficas de muchos-a-muchos

`Post` de un blog y un modelo `Video` pueden compartir una relación polimórfica con un modelo `Tag`. Usando una relación polimórfica de muchos-a-muchos te permite tener una única lista de etiquetas que son compartidas a través de posts y videos. Primero, vamos a examinar la estructura de tabla:

```php
posts
    id - integer
    name - string

videos
    id - integer
    name - string

tags
    id - integer
    name - string

taggables
    tag_id - integer
    taggable_id - integer
    taggable_type - string
```

Estructura del modelo

Seguidamente, estamos listos para definir las relaciones en el modelo. Ambos modelos `Post` y `Video` tendrán un método `tags` que ejecuta el método `morphToMany` en la clase base de

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
    * Get all of the tags for the post.
    */
    public function tags()
    {
        return $this->morphToMany('App\
    }
}
```

Definiendo el inverso de la relación

A continuación, en el modelo `Tag`, debes definir un método para cada uno de sus modelos relacionados. Así, para este ejemplo, definiremos un método `posts` y un método `videos`:

```php
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Tag extends Model
```

```
        * Get all of the posts that are ass.
        */
        public function posts()
        {
            return $this->morphedByMany('App
        }

        /**
        * Get all of the videos that are ass
        */
        public function videos()
        {
            return $this->morphedByMany('App
        }
    }
```

Obteniendo la relación

Una vez que tu tabla en la base de datos y modelos son definidos, puedes acceder las relaciones por medio de tus modelos. Por ejemplo, para acceder a todos los tags de un post, puedes usar la propiedad dinámica `tags` :

```php
$post = App\Post::find(1);

foreach ($post->tags as $tag) {
    //
}
```

También puedes obtener el propietario de una relación polimórfica desde el modelo polimórfico

son los métodos `posts` o `videos` en el modelo `Tag`. Así, accederemos a esos métodos como propiedades dinámicas:

```php
$tag = App\Tag::find(1);

foreach ($tag->videos as $video) {
    //
}
```

Tipos polimórficos personalizados

Por defecto, Laravel usará el nombre completo de clase para almacenar el tipo del modelo relacionado. Por ejemplo, dado el ejemplo uno-a-muchos de arriba donde un `Comment` puede pertenecer a un `Post` o a un `Video`, el `commentable_type` por defecto será `App\Post` o `App\Video`, respectivamente. Sin embargo, puedes querer desacoplar tu base de datos de la estructura interna de tu aplicación. En dicho caso, puedes definir un "mapa de morfología (morph map)" para indicarle a Eloquent que use un nombre personalizado para cada modelo en lugar del nombre de la clase:

```php
use Illuminate\Database\Eloquent\Relati

Relation::morphMap([
    'posts' => 'App\Post',
```

> **TIP**
>
> Al agregar un "morph map" a tu aplicación existente, cada valor de la columna de morfología `*_type` en tu base de datos que aún contenga una clase completamente calificada necesitará ser convertida a su nombre de "mapa".

Consultando relaciones

Ya que todos los tipos de relaciones Eloquent son definidas por medio de métodos, puedes ejecutar esos métodos para obtener una instancia de la relación sin ejecutar realmente las consultas de la relación. Además, todos los tipos de relaciones Eloquent también sirven como constructores de consultas, permitiendo que continues encadenando restricciones dentro de la consulta de la relación antes de ejecutar finalmente el SQL contra la base de datos.

Por ejemplo, imagina un sistema de blog en el cual un modelo `User` tiene muchos modelos `Post` asociados:

```php
<?php
```

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

class User extends Model
{
    /**
    * Get all of the posts for the user
    */
    public function posts()
    {
        return $this->hasMany('App\Post
    }
}
```

Puedes consultar la relación `posts` y agregar limitaciones a la relación de la siguiente forma:

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

$user->posts()->where('active', 1)->get(
```

Puedes usar cualquiera de los métodos de constructor de consultas y así que asegúrate de revisar la documentación del constructor de consultas para aprender sobre todos los métodos disponibles.

Encadenando cláusulas `orWhere` en relaciones

Como se demostró en el ejemplo superior, eres libre de agregar restriciones adicionales a las relaciones al momento de realizar peticiones. Sin embargo, ten

agrupadas lógicamente en el mismo nivel que la restricción de la relación:

```php
$user->posts()
        ->where('active', 1)
        ->orWhere('votes', '>=', 100)
        ->get();

// select * from posts
// where user_id = ? and active = 1 or
```

En la mayoria de los casos, probablemente pretendes usar grupos de restricciones para agrupar logicamente las comprobaciones condicionales entre parentisis:

```php
use Illuminate\Database\Eloquent\Builder

$user->posts()
        ->where(function (Builder $query
            return $query->where('active
                                ->orWhere('
        })
        ->get();

// select * from posts
// where user_id = ? and (active = 1 or
```

Métodos de relación Vs. propiedades dinámicas

acceder a la relación como si fuera una propiedad. Por ejemplo, continuando con el uso de nuestros modelos de ejemplo `User` y `Post`, podemos acceder a todos los posts de un usuario como sigue:

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

foreach ($user->posts as $post) {
    //
}
```

Las propiedades dinámicas son de "carga diferida (lazy loading)", lo que significa que cargarán solamente los datos de su relación cuando realmente accedas a ellas. Debido a esto, los desarrolladores con frecuencia usan carga previa (eager loading) para precargar las relaciones que ellos saben que serán accedidas después de cargar el modelo. La carga previa proporciona una reducción significativa en consultas SQL que deben ser ejecutadas para cargar las relaciones de un modelo.

Consultando la existencia de una relación

Cuando accedes a los registros de un modelo, puedes desear limitar sus resultados basados en la existencia de una relación. Por ejemplo, imagina que quieres obtener todos los posts de blog que tienen al menos un comentario. Para hacer eso, puedes pasar

```php
// Retrieve all posts that have at least
$posts = App\Post::has('comments')->get(
```

También puedes especificar un operador y la cantidad para personalizar aún más la consulta:

```php
// Retrieve all posts that have three or
$App/posts = Post::has('comments', '>=',
```

Las instrucciones `has` anidadas también pueden ser construidas usando la notación de "punto". Por ejemplo, puedes obtener todos los posts que tienen al menos un comentario con votos:

```php
// Retrieve posts that have at least one
$App/posts = Post::has('comments.votes')
```

Incluso si necesitas más potencia, puedes usar los métodos `whereHas` y `orWhereHas` para poner condiciones "where" en tus consultas `has`. Estos métodos permiten que agregues restricciones personalizadas a una restricción de relación, tal como verificar el contenido de un comentario:

```php
use Illuminate\Database\Eloquent\Builder

// Retrieve posts with at least one comm
```

```
})->get();

// Retrieve posts with at least ten comm
$posts = App\Post::whereHas('comments',
    $query->where('content', 'like', 'fo
}, '>=', 10)->get();
```

Consultando la ausencia de una relación

Al momento de acceder a los registros de un modelo, puedes desear limitar tus resultados en base a la ausencia de una relación. Por ejemplo, imagina que quieras obtener todos los posts de blogs que **no** tienen algún comentario. Para hacer eso, puedes pasar el nombre de la relación a los métodos `doesntHave` y `orDoesntHave`:

```php
$posts = App\Post::doesntHave('comments
```

Incluso si necesitas más potencia, puedes usar los métodos `whereDoesntHave` y `orWhereDoesntHave` para poner condiciones "where" en tus consultas `doesntHave`. Estos métodos permiten que agregues restricciones personalizadas a una restricción de relación, tal como verificar el contenido de un comentario:

```php
use Illuminate\Database\Eloquent\Builde
```

```php
    $query->where('content', 'like', 'f
})->get();
```

Puedes usar notación "de puntos" para ejecutar una consulta contra una relación anidada. Por ejemplo, la siguiente consulta entregará todos los posts con comentarios de autores que no están vetados:

```php
use Illuminate\Database\Eloquent\Builder

$posts = App\Post::whereDoesntHave('comm
    $query->where('banned', 1);
})->get();
```

Consultando relaciones polimorficas

Para consultar la existencia de relaciones `MorphTo`, puedes usar el método `whereHasMorph` y sus métodos correspondientes:

```php
use Illuminate\Database\Eloquent\Builder

// Retrieve comments associated to posts
$comments = App\Comment::whereHasMorph(
    'commentable',
    ['App\Post', 'App\Video'],
    function (Builder $query) {
        $query->where('title', 'like',
    }
)->get();
```

```
$comments = App\Comment::whereDoesntHave
    'commentable',
    'App\Post',
    function (Builder $query) {
        $query->where('title', 'like',
    }
)->get();
```

Puedes usar el parametro `$type` para agregar diferentes restricciones dependiendo del modelo relacionado:

```php
use Illuminate\Database\Eloquent\Builder

$comments = App\Comment::whereHasMorph(
    'commentable',
    ['App\Post', 'App\Video'],
    function (Builder $query, $type) {
        $query->where('title', 'like',

        if ($type === 'App\Post') {
            $query->orWhere('content',
        }
    }
)->get();
```

En lugar de pasar un arreglo de posibles modelos polimorficos, puedes proporcionar un `*` como comodín y dejar que Laravel retorne todos los posibles tipos polimorficos desde la base de datos. Laravel ejecutará una solicitud adicional para poder realizar esta operación:

```php
$comments = App\Comment::whereHasMorph(
    $query->where('title', 'like', 'foo%
})->get();
```

Contando modelos relacionados

Si quieres contar el número de resultados de una relación sin cargarlos realmente puedes usar el método `withCount`, el cual coloca una columna `{relation}_count` en tus modelos resultantes. Por ejemplo:

```php
$posts = App\Post::withCount('comments')

foreach ($posts as $post) {
    echo $post->comments_count;
}
```

Puedes agregar las "cuentas" para múltiples relaciones así como también agregar restricciones a las consultas:

```php
$posts = Post::withCount(['votes', 'comm
    $query->where('content', 'like', 'fo
}])->get();

echo $posts[0]->votes_count;
echo $posts[0]->comments_count;
```

en la misma relación:

```php
use Illuminate\Database\Eloquent\Builder

$posts = App/post::withCount([
    'comments',
    'comments as pending_comments_count
        $query->where('approved', false)
    },
])->get();

echo $posts[0]->comments_count;

echo $posts[0]->pending_comments_count;
```

Si estás combinando `withCount` con una instrucción `select`, asegúrate de llamar a `withCount` después del método `select`:

```php
$posts = App\Post::select(['title', 'bo

echo $posts[0]->title;
echo $posts[0]->body;
echo $posts[0]->comments_count;
```

Además, utilizando el método `loadCount`, puedes cargar un conteo de la relación después de que el modelo padre haya sido obtenido:

```php
$book = App\Book::first();
```

Si necesitas establecer restricciones adicionales de consulta en la consulta de carga previa, puedes pasar un arreglo clave por las relaciones que deseas cargar. Los valores del arreglo deben ser instancias de `Closure` que reciben la instancia del generador de consulta:

```php
$book->loadCount(['reviews' => function
    $query->where('rating', 5);
}])
```

Carga previa (eager loading)

Al momento de acceder a las relaciones Eloquent como propiedades, los datos de la relación son ”cargados diferidamente (lazy loading)”. Esto significa que los datos de la relación no son cargados realmente hasta que primero accedas a la propiedad. Sin embargo, Eloquent puede ”cargar previamente (eager loading)” las relaciones al mismo tiempo que consultas el modelo padre. La carga previa alivia el problema de la consulta N + 1. Para ilustrar el problema de la consulta N + 1, considera un modelo `Book` que está relacionado a `Author` :

```php
<?php

namespace App;
```

```
class Book extends Model
{
    /**
    * Get the author that wrote the bool
    */
    public function author()
    {
        return $this->belongsTo('App\Aut
    }
}
```

Ahora, vamos a obtener todos los libros y sus autores:

```php
$books = App\Book::all();

foreach ($books as $book) {
    echo $book->author->name;
}
```

Este ciclo ejecutará una consulta para obtener todos los libros en la tabla, despues otra consulta para cada libro para obtener el autor. Así, si tenemos 25 libros, este ciclo debería ejecutar 26 consultas: 1 para el libro original y 25 consultas adicionales para obtener el autor de cada libro.

Afortunadamente, podemos usar la carga previa para reducir esta operación a solo 2 consultas. Al momento de consultar, puedes especificar cuáles relaciones deberían ser precargadas usando el

```php
$books = App\Book::with('author')->get(

foreach ($books as $book) {
    echo $book->author->name;
}
```

Para esta operación, solo dos consultas serán ejecutadas:

```php
select * from books

select * from authors where id in (1, 2,
```

Carga previa de múltiples relaciones

Algunas veces puedes necesitar la carga previa de varias relaciones diferentes en una operación única. Para hacer eso, pasa sólo los argumentos adicionales al método `with` :

```php
$books = App\Book::with(['author', 'pub
```

Carga previa anidada

Para precargar relaciones anidadas, puedes usar la sintaxis de "punto". Por ejemplo, vamos a precargar todos los autores de los libros y todos los contactos personales del autor en una instrucción de Eloquent:

Eager Load anidado de relaciones `morphTo`

Si te gustaría hacer eager load de relaciones `morphTo`, así como de relaciones anidadas en varias entidades que podrían ser retornadas por dicha relación, puedes usar el método `with` en combinación con el método `morphWith` de la relación `morphTo`. Para ayudarte a ilustrar este método, vamos a considerar el siguiente método:

```php
<?php

use Illuminate\Database\Eloquent\Model;

class ActivityFeed extends Model
{
    /**
    * Get the parent of the activity fe
    */
    public function parentable()
    {
        return $this->morphTo();
    }
}
```

En este ejemplo, vamos a asumir que los modelos `Èvent`, `Photo` y `Post` podrían crear moelos `ActivityFeed`. Adicionalmente, vamos a asumir que los modelos `Event` pertenecen a una modelo `Calendar`, que los modelos `Photo` están

Usando estas definiciones de modelos y relaciones, podríamos retornar instancias del modelo `ActivityFeed` y hacer eager load de todos los modelos `parentable` y sus respectivas relaciones anidadas:

```php
use Illuminate\Database\Eloquent\Relati

$activities = ActivityFeed::query()
    ->with(['parentable' => function (M
        $morphTo->morphWith([
            Event::class => ['calendar'
            Photo::class => ['tags'],
            Post::class => ['author'],
        ]);
    }])->get();
```

Cargando previamente columnas específicas

No siempre necesitas todas las columna de las relaciones que estás obteniendo. Por esta razón, Eloquent te permite que especificar cuáles columnas de la relación te gustaría obtener:

```php
$books = App\Book::with('author:id,name
```

> **Nota**
>
> Al momento de usar esta característica,

Para relaciones "tiene muchos" necesitas especificar tanto `id` como `foreign_key`

```php
$books = App\Book::with('chapter:id,bool
```

> **Nota**
>
> Al usar esta caracteristica, siempre debes incluir la columna `id` y cualquier columna de clave foranea relevante en la lista de columnas que deseas retornar.

Carga previa por defecto

Algunas veces vas a querer cargar siempre algunas relaciones al retornar un modelo. Para lograr esto, puedes definir una propiedad `$with` en el modelo:

```php
<?php

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

class Book extends Model
{
    /**
    * Always load the related author whe
    * The relationships that should alwa
```

```
    */
    protected $with = ['author'];

    /**
    * Get the author that wrote the bool
    */
    public function author()
    {
        return $this->belongsTo('App\Aut
    }
}
```

Si te gustaria remover un elemento de la propiedad `$with` para una sola petición, puedes usar el método `without` :

```php
$books = App\Book::without('author')->ge
```

Restringiendo cargas previas

Algunas veces puedes desear cargar previamente una relación, pero también especificar condiciones de consulta para la consulta de carga previa. Aquí está un ejemplo:

```php
use Illuminate\Database\Eloquent\Builde

$users = App\User::with(['posts' => func
    $query->where('title', 'like', '%fi
}])->get();
```

la palabra `first`. Puedes ejecutar otros métodos del constructor de consulta para personalizar más la operación de carga previa:

```php
use Illuminate\Database\Eloquent\Builder

$users = App\User::with(['posts' => func
    $query->orderBy('created_at', 'desc
}])->get();
```

> **Nota**
>
> Los métodos del constructor de consultas `limit` y `take` no se pueden usar al restringir las cargas previas.

Carga previa diferida (lazy eager loading)

Algunas veces puedes necesitar precargar una relación después de que el modelo padre ya ha sido obtenido. Por ejemplo, esto puede ser útil si necesitas decidir dinámicamente si se cargan modelos relacionados:

```php
$books = App\Book::all();

if ($someCondition) {
    $books->load('author', 'publisher');
```

Si necesitas establecer restricciones de consultas adicionales en la consulta de carga previa, puedes pasar un arreglo clave / valor con las relaciones que deseas cargar. Los valores del arreglo deberían ser instancias de `Closure`, las cuales reciben la instancia de consulta:

```php
use Illuminate\Database\Eloquent\Builder

$books->load(['author' => function ($que
    $query->orderBy('published_date', 'a
}]);
```

Para cargar una relación solo cuando aún no se ha cargado, usa el método `loadMissing`:

```php
public function format(Book $book)
{
    $book->loadMissing('author');

    return [
        'name' => $book->name,
        'author' => $book->author->name,
    ];
}
```

Carga previa diferida anidada y `morphTo`

Si deseas cargar previamente una relación `morphTo`, así como relaciones anidadas en las

Este método acepta el nombre de la relación `morphTo` como su primer argumento, y un arreglo de pares modelo / relación como su segundo argumento. Para ayudar a ilustrar este método, consideremos el siguiente modelo:

```php
<?php

use Illuminate\Database\Eloquent\Model;

class ActivityFeed extends Model
{
    /**
    * Get the parent of the activity fe
    */
    public function parentable()
    {
        return $this->morphTo();
    }
}
```

En este ejemplo, asumamos que los modelos `Event`, `Photo` y `Post` pueden crear modelos `ActivityFeed`. Además, supongamos que los modelos `Event` pertenecen a un modelo `Calendar`, los modelos `Photo` están asociados con los modelos `Tag` y los modelos `Post` pertenecen a un modelo `Author`.

Usando estas definiciones y relaciones de modelo, podemos recuperar instancias de modelo

relaciones anidadas:

```php
$activities = ActivityFeed::with('parent
    ->get()
    ->loadMorph('parentable', [
        Event::class => ['calendar'],
        Photo::class => ['tags'],
        Post::class => ['author'],
    ]);
```

Insertando y actualizando modelos relacionados

El método save

Eloquent proporciona métodos convenientes para agregar nuevos modelos a las relaciones. Por ejemplo, quizá necesites insertar un nuevo `Comment` para un modelo `Post`. En lugar de establecer manualmente el atributo `post_id` en el `Comment`, puedes insertar el `Comment` directamente con el método `save` de la relación:

```php
$comment = new App\Comment(['message' =>

$post = App\Post::find(1);

$post->comments()->save($comment);
```

ejecutamos el método `comments` para obtener una instancia de la relación. El método `save` automáticamente agregará el valor `post_id` apropiado al nuevo modelo `Comment` .

Si necesitas guardar múltiples modelos relacionados, puedes usar el método `saveMany` :

```php
$post = App\Post::find(1);

$post->comments()->saveMany([
    new App\Comment(['message' => 'A new
    new App\Comment(['message' => 'Anoth
]);
```

Guardando modelos y relaciones recursivamente

SI quieres hacer `save` a tu modelo y a todas sus relaciones asociadas, puedes usar el método `push` :

```php
$post = App\Post::find(1);

$post->comments[0]->message = 'Message',
$post->comments[0]->author->name = 'Auth

$post->push();
```

El método create

En adición a los métodos `save` y `saveMany` ,

inserta dentro de la base de datos. Otra vez, la diferencia entre `save` y `create` es que `save` acepta una instancia de modelo Eloquent llena mientras `create` acepta un `array` PHP plano:

```php
$post = App\Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
]);
```

> **TIP**
>
> Antes de usar el método `create`, asegurate de revisar la documentación sobre la asignación masiva de atributos.

Puedes usar el método `createMany` para crear múltiples modelos relacionados:

```php
$post = App\Post::find(1);

$post->comments()->createMany([
    [
        'message' => 'A new comment.',
    ],
    [
        'message' => 'Another new commer
    ],
]);
```

`updateOrCreate` para crear y actualizar modelos en relaciones.

Actualizar relación pertenece a (belongsTo)

Al momento de actualizar una relación `belongsTo`, puedes usar el método `associate`. Este método establecerá la clave foránea en el modelo hijo:

```php
$account = App\Account::find(10);

$user->account()->associate($account);

$user->save();
```

Al momento de eliminar una relación `belongsTo`, puedes usar el método `dissociate`. Este método establecerá la clave foránea de la relación a `null`:

```php
$user->account()->dissociate();

$user->save();
```

Modelos predeterminados

Las relaciones `belongsTo`, `hasOne`, `hasOneThrough` y `morphOne` te permiten definir un modelo predeterminado que se devolverá si la relación dada es `null`. A este patrón se le conoce

código. En el ejemplo siguiente, la relación `user` devolverá un modelo `App\User` vacío si no hay un `user` adjunto a la publicación:

```php
/**
* Get the author of the post.
*/
public function user()
{
    return $this->belongsTo('App\User')
}
```

Para rellenar el modelo predeterminado con atributos, puedes pasar un arreglo o Closure al método `withDefault` :

```php
/**
* Get the author of the post.
*/
public function user()
{
    return $this->belongsTo('App\User')
        'name' => 'Guest Author',
    ]);
}

/**
* Get the author of the post.
*/
public function user()
{
```

```
        });
    }
```

Relaciones muchos a muchos

Adjuntando (attach) / Quitando (detach)

Eloquent también proporciona unos cuantas métodos helper para hacer que el trabajo con los modelos relacionados sea más conveniente. Por ejemplo, vamos a imaginar que un usuario tiene muchos roles y un rol puede tener muchos usuarios. Para adjuntar un rol a un usuario insertando un registro en la tabla intermedia que vincula los modelos, usa el método `attach` :

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

$user->roles()->attach($roleId);
```

Al momento de adjuntar una relación a un modelo, también puedes pasar un arreglo de datos adicionales para ser insertados dentro de la tabla intermedia:

```php
$user->roles()->attach($roleId, ['expire
```

Algunas veces puede ser necesario quitar un rol de un usuario. Para remover un registro de una relación

la tabla intermedia; sin embargo, ambos modelos permanecerán en la base de datos:

```php
// Detach a single role from the user..
$user->roles()->detach($roleId);

// Detach all roles from the user...
$user->roles()->detach();
```

Por conveniencia, `attach` y `detach` también aceptan arreglos de IDs como entrada:

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

$user->roles()->detach([1, 2, 3]);

$user->roles()->attach([
    1 => ['expires' => $expires],
    2 => ['expires' => $expires],
]);
```

Sincronizando asociaciones

También puedes usar el método `sync` para construir asociaciones muchos-a-muchos. El método `sync` acepta un arreglo de IDs para colocar en la tabla intermedia. Algunos IDs que no estén en el arreglo dado serán removidos de la tabla intermedia. Por tanto, después que esta operación se complete, solamente los IDs en el arreglo dado existirán en la

```php
$user->roles()->sync([1, 2, 3]);
```

También puedes pasar valores adicionales de tabla intermedia con los IDs:

```php
$user->roles()->sync([1 => ['expires' =>
```

Si no quieres desatar IDs existentes, puedes usar el método `syncWithoutDetaching`:

```php
$user->roles()->syncWithoutDetaching([1,
```

Alternar asociaciones

La relación de muchos-a-muchos también proporciona un método `toggle` el cual "alterna" el estado adjunto de los IDs dados. Si el ID está actualmente adjuntado, será removido. De igual forma, si está actualmente removido, será adjuntado:

```php
$user->roles()->toggle([1, 2, 3]);
```

Guardando datos adicionales en una tabla pivote

Al momento de trabajar con una relación de muchos-a-muchos, el método `save` acepta un arreglo de atributos adicionales de tabla intermedia como su segundo argumento:

Actualizando un registro en una tabla pivote

Si necesitas actualizar una fila existente en tu tabla pivote, puedes usar el método `updateExistingPivot`. Este método acepta la clave foránea del registro pivote y un arreglo de atributos para actualizar:

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

$user->roles()->updateExistingPivot($ro
```

Tocando marcas de tiempo del padre

Cuando un modelo `belongsTo` o `belongsToMany` a otro modelo, tal como un `Comment` el cual pertenece a un `Post`, algunas veces es útil actualizar la marca de tiempo del padre cuando el modelo hijo es actualizado. Por ejemplo, cuando un modelo `Comment` es actualizado, puedes querer "tocar" automáticamente la marca de tiempo `updated_at` del `Post` que lo posee. Eloquent hace esto fácil. Simplemente agrega una propiedad `touches` conteniendo los nombres de las relaciones al modelo hijo:

```php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
    * All of the relationships to be tou
    *
    * @var array
    */
    protected $touches = ['post'];

    /**
    * Get the post that the comment belo
    */
    public function post()
    {
        return $this->belongsTo('App\Pos
    }
}
```

Ahora, cuando actualices un `Comment`, el `Post` que lo posee tendrá su columna `updated_at` actualizada también, haciéndolo más conveniente para saber cuándo invalidar una caché del modelo `Post`:

```php
$comment = App\Comment::find(1);

$comment->text = 'Edit to this comment!
```

← Eloquent: Primeros Pasos Eloquent: Colecciones →