## Uno a muchos (one to many)

Un ejemplo de una relación uno-a-muchos es un artículo de un Blog que "tiene varios" comentarios. Podemos modelar esta relación así:

```
class Post extends Model {

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

}
```

Ahora podemos acceder a los comentarios del post a través de la propiedad dinámica:

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

Si necesitas agregar algunas limitaciones a los comentarios que obtuviste, puedes llamar al método `comments` y continuar encadenándole condiciones:

```
$comments = Post::find(1)->comments()->where('title',
'=', 'foo')->first();
```

De nuevo, se puede reemplazar la clave ajena convencional pasando esta como segundo argumento del método `hasMany`. Y, como en la relación `hasOne`, la columna local puede ser además especificada:

```
return $this->hasMany('App\Comment', 'foreign_key');

return $this->hasMany('App\Comment', 'foreign_key',
'local_key');
```

### Definir la inversa de una relación

Para definir la inversa de la relación del modelo `Comment`, utilizaremos el método `belongsTo`:

```
class Comment extends Model {

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

```
}
```

## Muchos a muchos (many to many)

Las relaciones muchos-a-muchos son un tipo de relación más compleja. Un ejemplo de esta relación es un usuario con varios roles, donde los roles son además compartidos por otros usuarios. Por ejemplo, muchos usuarios pueden poseer el rol de "Admin". Tres tablas en la base de datos harían falta para esta relación: `users`, `roles`, y `role_user`. La tabla `role_user` deriva del orden alfabético de los nombres de los modelos relacionados, y debe contener las columnas `user_id` y `rol_id`.

Podemos definir una relación muchos-a-muchos utilizando el método `belongsToMany`:

```
class User extends Model {

    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }

}
```

Ahora podemos recuperar los roles a través del modelo `User`:

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

Si quieres utilizar un nombre no convencional para tu tabla pivote, puedes pasarlo como segundo argumento al método `belongsToMany`:

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

También puedes reemplazar las keys asociadas por defecto:

```
return $this->belongsToMany('App\Role', 'user_roles',
'user_id', 'foo_id');
```

De hecho, también puedes definir la inversa de la relación en el modelo `Role`:

```
```

```
class Role extends Model {

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

}
```

## Muchos a través de (has many through)

La relación "tiene muchos a través de" proporciona un atajo conveniente para acceder a las relaciones distantes mediante una relación intermedia. Pro ejemplo, un modelo `Country` podría tener muchos `Post` a través de un modelo `User`. Las tablas para esta relación tendrían este aspecto:

```
countries
    id - integer
    name - string

users
    id - integer
    country_id - integer
    name - string

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

Aunque la tabla `posts` no contenga una columna `country_id`, la relación `hasManyThrough` permitirá acceder a los post por países (countries) vía `$country->posts`. Vamos a definir la relación:

```
class Country extends Model {

    public function posts()
    {
        return $this->hasManyThrough('App\Post',
'App\User');
    }

}
```

Puedes también especificar de forma manual las claves (llaves) de la relación pasándolas como tercer y cuarto argumentos al método:

```
class Country extends Model {

    public function posts()
    {
        return $this->hasManyThrough('App\Post',
'App\User', 'country_id', 'user_id');
    }

}
```

## Relaciones polimórficas (polymorphic relations)

Las relaciones polimórficas permiten a un modelo pertenecer a mas de un modelo, en una asociación sencilla. Por ejemplo, en un mismo modelo Photo puedes tener fotos del modelo Staff y del modelo Order. Definiríamos esta relación del siguiente modo:

```
class Photo extends Model {

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

}

class Staff extends Model {

    public function photos()
    {
        return $this->morphMany('App\Photo',
'imageable');
    }

}

class Order extends Model {

    public function photos()
    {
        return $this->morphMany('App\Photo',
'imageable');
    }
```

```
}
```

### Obtener una relación polimórfica

Ahora, podemos recuperar las fotos de cada miembro del staff u order:

```
$staff = Staff::find(1);

foreach ($staff->photos as $photo)
{
    //
}
```

### Recuperando el propietario de una relación polimórfica

Sin embargo, la verdadera magia "polimórfica" aparece cuando se accede al staff u order desde el modelo `Photo`:

```
$photo = Photo::find(1);

$imageable = $photo->imageable;
```

La relación `imageable` en el modelo `Photo` retornará una instancia `Staff` u `Order`, dependiendo de qué tipo de pertenece esa foto.

### Estructura de tabla para una relación polimórfica

Para ayudar a comprender cómo funciona esto, vamos a explorar la estructura de base de datos para una relación polimórfica:

```
staff
    id - integer
    name - string

orders
    id - integer
    price - integer

photos
    id - integer
    path - string
    imageable_id - integer
    imageable_type - string
```

Los campos que nos importan aquí son el `imageable_id` y `imageable_type` en la tabla `photos`. El ID contendrá el valor ID de, en este ejemplo, el propietario staff u order, mientras el tipo contendrá el nombre de la clase modelo propietaria. Esto es lo que permite al ORM determinar que tipo de modelo propietario ha de retornar cuando se accede a la relación `imageable`.

## Relación polimórfica de muchos a muchos

### Estructura de tabla para una relación polimórfica de muchos-a-muchos

Además de las relaciones polimórficas habituales, también se pueden especificar relaciones polimórficas "many-to-many" (muchos-a-muchos). Por ejemplo, el modelo de un `Post` de un blog y un `Video` podrían compartir una relación polimórfica con un modelo de `Tag`. Primero, examinemos la estructura de la tabla:

```
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
```

A continuación, estamos listos para configurar las relaciones en el modelo. Ambos modelos `Post` y `Video` tendrán una relación `morphToMany` vía el método `tags`:

```
class Post extends Model {

    public function tags()
    {
        return $this->morphToMany('App\Tag',
```

```
'taggable');
    }

}
```

El modelo `Tags` puede definir un método para cada una de sus relaciones:

```
class Tag extends Model {

    public function posts()
    {
        return $this->morphedByMany('App\Post',
'taggable');
    }

    public function videos()
    {
        return $this->morphedByMany('App\Video',
'taggable');
    }

}
```