Course - Laravel Framework

Quan hệ ORM

Các bảng cơ sở dữ liệu thường liên quan đến nhau. Ví dụ: một bài đăng trên blog có thể có nhiều bình luận hoặc một đơn hàng có thể liên quan đến người dùng đã đặt nó. Eloquent giúp việc quản lý và làm việc với các mối quan hệ này trở nên dễ dàng và hỗ trợ nhiều mối quan hệ phổ biến

Tags: relationship orm, quan he du lieu orm, laravel

Giới thiệu

Các bảng cơ sở dữ liệu thường liên quan đến nhau. Ví dụ: một bài đăng trên blog có thể có nhiều bình luận hoặc một đơn hàng có thể liên quan đến người dùng đã đặt nó. Eloquent giúp việc quản lý và làm việc với các mối quan hệ này trở nên dễ dàng và hỗ trợ nhiều mối quan hệ phổ biến:

- Một đối một
- Một đối nhiều
- Nhiều đối nhiều
- Một qua một
- Một qua nhiều
- Một đối một (đa hình)
- Một đối nhiều (đa hình)
- Nhiều đối nhiều (đa hình)

Tạo các mối quan hệ

Mối quan hệ Eloquent được khai nhận dưới dạng các phương thức trên các class Eloquent model của bạn. Vì mối quan hệ cũng đóng vai trò là query builder mạnh mẽ, việc xác định mối quan hệ dưới dạng phương thức cung cấp khả năng truy vấn và xâu chuỗi phương thức mạnh mẽ. Ví dụ: chúng ta có thể xâu chuỗi các ràng buộc truy vấn bổ sung trên mối quan hệ bài đăng này:

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

Tuy nhiên, trước khi đi quá sâu vào việc sử dụng các mối quan hệ, chúng ta hãy tìm hiểu cách xác định từng loại mối quan hệ được hỗ trợ bởi Eloquent.

Một đối một

Mối quan hệ một đối một là một kiểu quan hệ cơ sở dữ liệu rất cơ bản. Ví dụ: model `User` có thể được liên kết với một model `Phone`. Để xác định mối quan hệ này, chúng ta sẽ đặt một phương thức `phone` trên model `User`. Phương thức `phone` sẽ gọi phương thức `hasOne` và trả về kết quả của nó. Phương thức `hasOne` có sẵn cho model của bạn thông qua class cơ sở `Illuminate\Database\Eloquent\Model` của model:

```
<?php
```

```php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
  /**
   * Get the phone associated with the user.
   */
  public function phone()
  {
    return $this->hasOne(Phone::class);
  }
}
```

Đối số đầu tiên được truyền cho phương thức `hasOne` là tên của class model liên quan. Khi mối quan hệ được xác định, chúng tôi có thể truy xuất record liên quan bằng cách sử dụng các thuộc tính động của Eloquent. Thuộc tính động cho phép bạn truy cập các phương thức quan hệ như thể chúng là các thuộc tính được xác định trên mô hình:

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

Eloquent xác định khóa ngoại của mối quan hệ dựa trên tên của model mẹ. Trong trường hợp này, model `Phone` sẽ tự động được giả định có khóa ngoại `user_id`. Nếu bạn muốn ghi đè quy ước này, bạn có thể truyền đối số thứ hai vào phương thức `hasOne`:

```php
return $this->hasOne(Phone::class, 'foreign_key');
```

Ngoài ra, Eloquent giả định rằng khóa ngoại phải có giá trị khớp với cột khóa chính của model cha. Nói cách khác, Eloquent sẽ tìm kiếm giá trị của cột `id` của người dùng trong cột `user_id` của record `Phone`. Nếu bạn muốn mối quan hệ sử dụng giá trị khóa chính khác với `id` hoặc thuộc tính `$primaryKey` của model của bạn, thì bạn có thể truyền đối số thứ ba cho phương thức `hasOne`:

```php
return $this->hasOne(Phone::class, 'foreign_key', 'local_key');
```

Phần nghịch đảo của mối quan hệ

Có thể, chúng ta thường truy cập model `Phone` từ model `User` của chúng ta. Vì thế tiếp theo, hãy xác định mối quan hệ trên model `Phone` mà cho phép chúng ta truy cập vào người dùng sở hữu điện thoại. Chúng ta có thể xác định phần nghịch đảo của một mối quan hệ `hasOne` bằng cách sử dụng phương thức `belongsTo`:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model
{
  /**
   * Get the user that owns the phone.
   */
  public function user()
  {
    return $this->belongsTo(User::class);
  }
}
```

Khi gọi phương thức `user`, Eloquent sẽ cố gắng tìm model `User` có `id` khớp với cột `user_id` trên model `Phone`.

Eloquent xác định tên khóa ngoại bằng cách kiểm tra tên của phương thức quan hệ và thêm tên phương thức bằng `_id`. Vì vậy, trong trường hợp này, Eloquent giả định rằng model Phone có cột `user_id`. Tuy nhiên, nếu khóa ngoại trên model `Phone` không phải là `user_id`, bạn có thể truyền tên khóa tùy chỉnh vào đối số thứ hai của phương thức `belongsTo`:

```
/**
 * Get the user that owns the phone.
 */
public function user()
{
```

```
  return $this->belongsTo(User::class, 'foreign_key');
}
```

Nếu model mẹ không sử dụng `id` làm khóa chính của nó hoặc bạn muốn tìm model được liên kết bằng cách sử dụng một cột khác, bạn có thể truyền thêm đối số thứ ba cho phương thức `belongsTo` để chỉ định khóa quan hệ do bạn chọn của bảng mẹ:

```
/**
 * Get the user that owns the phone.
 */
public function user()
{
  return $this->belongsTo(User::class, 'foreign_key', 'owner_key');
}
```

Một đối nhiều

Mối quan hệ một đối nhiều được sử dụng để xác định các mối quan hệ trong đó một model thống nhất là model mẹ với một hoặc nhiều model con. Ví dụ, một bài đăng trên blog có thể có vô số bình luận. Giống như tất cả các mối quan hệ Eloquent khác, các mối quan hệ một đối nhiều được xác định bằng cách tạo một phương thức trên Eloquent model của bạn:

```
<?php

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

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

```
}
```

Hãy nhớ rằng, Eloquent sẽ tự động xác định cột khóa ngoại thích hợp cho model `Comment`. Theo quy ước, Eloquent sẽ lấy tên "trường hợp con rắn" của model mẹ và tiếp nối nó với `_id`. Vì vậy, trong ví dụ này, Eloquent sẽ giả sử cột khóa ngoại trên model `Comment` là `post_id`.

Khi phương thức mối quan hệ đã được xác định, chúng ta có thể truy cập vào tập hợp bình luận có liên quan bằng cách truy cập thuộc tính `comments`. Hãy nhớ rằng, vì Eloquent cung cấp "thuộc tính quan hệ động", chúng ta có thể truy cập các phương thức quan hệ như thể chúng được định nghĩa là thuộc tính trên model:

```
use App\Models\Post;

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

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

Vì tất cả các mối quan hệ cũng đóng vai trò là query builder, bạn có thể thêm các ràng buộc khác vào truy vấn mối quan hệ bằng cách gọi phương thức `comments` và tiếp tục xâu chuỗi các điều kiện vào truy vấn:

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

Giống như phương thức `hasOne`, bạn cũng có thể ghi đè các khóa ngoại và khóa cục bộ bằng cách chuyển các đối số bổ sung cho phương thức `hasMany`:

```
return $this->hasMany(Comment::class, 'foreign_key');

return $this->hasMany(Comment::class, 'foreign_key', 'local_key');
```

Phần nghịch đảo một đối nhiều

Bây giờ chúng ta có thể truy cập tất cả các nhận xét của một bài đăng, hãy xác định mối quan hệ để cho phép một nhận xét truy cập vào bài đăng chính của nó. Để xác định phần nghịch đảo của một mối quan hệ `hasMany`, hãy tạo một phương thức quan hệ trên model con gọi phương thức `belongsTo`:

```php
<?php

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

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

Khi mối quan hệ đã được tạo, chúng ta có thể truy xuất bài đăng của chính nhận xét bằng cách truy cập vào "thuộc tính quan hệ động" `post` như sau:

```php
use App\Models\Comment;

$comment = Comment::find(1);

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

Trong ví dụ trên, Eloquent sẽ cố gắng tìm một model `Post` có `id` khớp với cột `post_id` trên model `Comment`.

Eloquent chọn tên khóa ngoại mặc định bằng cách kiểm tra tên của phương thức quan hệ và thêm sau tên phương thức với dấu gạch dưới _ bằng tên của cột khóa chính của model mẹ. Vì vậy, trong ví dụ này, Eloquent sẽ giả sử khóa ngoại của model `Post` trên bảng `comments` là `post_id`.

Tuy nhiên, nếu khóa ngoại cho mối quan hệ của bạn không tuân theo các quy ước này, bạn có thể truyền tên khóa ngoại tự chọn làm đối số thứ hai cho phương thức `belongTo`:

```
/**
 * Get the post that owns the comment.
 */
public function post()
{
  return $this->belongsTo(Post::class, 'foreign_key');
}
```

Nếu model mẹ của bạn không sử dụng `id` làm khóa chính của nó hoặc bạn muốn tìm model được liên kết bằng cách sử dụng một cột khác, bạn có thể truyền đối số thứ ba đến phương thức `belongsTo` chỉ định khóa tùy chỉnh của bảng mẹ của bạn:

```
/**
 * Get the post that owns the comment.
 */
public function post()
{
  return $this->belongsTo(Post::class, 'foreign_key', 'owner_key');
}
```

Các mô hình mặc định

Các mối quan hệ `belongsTo`, `hasOne`, `hasOneThrough` và `morphOne` cho phép bạn xác định một model mặc định sẽ được trả về nếu mối quan hệ đã cho là `null`. Mẫu pattern này thường được gọi là pattern Null Object và có thể giúp loại bỏ các kiểm chứng có điều kiện trong mã của bạn. Trong ví dụ sau, mối quan hệ `user` sẽ trả về model trống `App\Models\User` trống nếu không có người dùng nào được đính kèm với `Post` model:

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

```
    return $this->belongsTo(User::class)->withDefault();
}
```

Để thao tác model mặc định với các thuộc tính, bạn có thể chuyển một mảng hoặc hàm xử lý vào phương thức `withDefault`:

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

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

Truy vấn quan hệ *Belongs To*

Khi truy vấn phần con của mối quan hệ *"belongs to"*, bạn có thể xây dựng mệnh đề `where` theo cách thủ công để truy xuất các Eloquent model tương ứng:

```
use App\Models\Post;

$posts = Post::where('user_id', $user->id)->get();
```

Tuy nhiên, bạn có thể sẽ thấy thuận tiện hơn khi sử dụng phương thức `whereBelongsTo`,

phương thức này sẽ tự động xác định mối quan hệvà khóa ngoại thích hợp cho model đã cho:

```
$posts = Post::whereBelongsTo($user)->get();
```

Theo mặc định, Laravel sẽ xác định mối quan hệ được liên kết với model đã cho dựa trên tên class của model; tuy nhiên, bạn có thể chỉ định tên mối quan hệ theo cách thủ công bằng cách cung cấp nó làm đối số thứ hai cho phương thức `whereBelongsTo`:

```
$posts = Post::whereBelongsTo($user, 'author')->get();
```

Có một của nhiều (one of many)

Đôi khi một model có thể có nhiều model liên quan, nhưng bạn muốn dễ dàng truy xuất model liên quan "mới nhất" hoặc "cũ nhất" của mối quan hệ. Ví dụ: model `User` có thể liên quan đến nhiều model `Order`, nhưng bạn muốn xác định một cách thuận tiện để tương tác với đơn hàng gần đây nhất mà người dùng đã đặt. Bạn có thể thực hiện điều này bằng cách sử dụng kiểu quan hệ `hasOne` kết hợp với các phương thức `ofMany`:

```
/**
 * Get the user's most recent order.
 */
public function latestOrder()
{
  return $this->hasOne(Order::class)->latestOfMany();
}
```

Tương tự như vậy, bạn có thể xác định một phương thức để truy xuất model liên quan "cũ nhất" hoặc đầu tiên của mối quan hệ:

```
/**
 * Get the user's oldest order.
 */
public function oldestOrder()
{
  return $this->hasOne(Order::class)->oldestOfMany();
```

```
}
```

Theo mặc định, các phương thức `latestOfMany` mới nhất và `oldestOfMany` cũ nhất sẽ truy xuất model có liên quan mới nhất hoặc cũ nhất dựa trên khóa chính của mô hình, khóa này phải có thể xếp thứ tự được. Tuy nhiên, đôi khi bạn có thể muốn truy xuất một model từ một mối quan hệ lớn hơn bằng cách sử dụng một tiêu chí sắp xếp thứ tự khác.

Ví dụ: khi sử dụng phương thức `ofMany`, bạn có thể truy xuất đơn đặt hàng đắt nhất của người dùng. Phương thức `ofMany` chấp nhận cột có thể sắp xếp thứ tự và hàm tổng hợp (min hoặc max) làm đối số của nó khi truy vấn model liên quan:

```
/**
 * Get the user's largest order.
 */
public function largestOrder()
{
  return $this->hasOne(Order::class)->ofMany('price', 'max');
}
```

> **Chú ý:** Vì PostgreSQL không hỗ trợ thực thi hàm `MAX` đối với các cột UUID, nên hiện tại không thể sử dụng mối quan hệ một đối nhiều kết hợp với các cột UUID của PostgreSQL.

Mối quan hệ *một trong nhiều* nâng cao

Bạn có thể xây dựng các mối quan hệ "một trong nhiều" nâng cao hơn. Ví dụ: Một model `Product` có thể có nhiều model `Price` liên quan được giữ lại trong hệ thống ngay cả sau khi giá mới được công bố. Ngoài ra, dữ liệu giá mới cho sản phẩm có thể được xuất bản trước để có hiệu lực vào một ngày trong tương lai thông qua cột `published_up`.

Vì vậy, tóm lại, chúng tôi cần truy xuất giá được công bố mới nhất mà ngày công bố không phải là trong tương lai. Ngoài ra, nếu hai giá có cùng ngày xuất bản, chúng tôi sẽ ưu tiên giá có *ID* lớn nhất. Để thực hiện điều này, chúng ta phải truyền một mảng cho phương thức `ofMany` có chứa các cột có thể sắp xếp thứ tự để xác định giá mới nhất. Ngoài ra, một hàm xử lý sẽ được cung cấp làm đối số thứ hai cho phương thức `ofMany`. Hàm này sẽ chịu trách nhiệm thêm các ràng buộc ngày xuất bản bổ sung vào truy vấn mối quan hệ:

```
/**
```

```php
 * Get the current pricing for the product.
 */
public function currentPricing()
{
  return $this->hasOne(Price::class)->ofMany([
    'published_at' => 'max',
    'id' => 'max',
  ], function ($query) {
    $query->where('published_at', '<', now());
  });
}
```

Một qua một

Mối quan hệ một qua một "has-one-through" xác định mối quan hệ 1-1 với một model khác. Tuy nhiên, mối quan hệ này chỉ ra rằng một model khớp với một model khác bằng một model thứ ba.

Ví dụ: trong ứng dụng cửa hàng sửa chữa xe, mỗi model `Mechinic` có thể được liên kết với một model `Car` và mỗi model `Car` có thể được liên kết với một model `Owner`. Mặc dù thợ máy và chủ sở hữu không có mối quan hệ trực tiếp trong cơ sở dữ liệu, nhưng thợ máy có thể truy cập chủ sở hữu thông qua model `Car`. Hãy xem các bảng cần thiết để xác định mối quan hệ này:

```
mechanics
  id - integer
  name - string

cars
  id - integer
  model - string
  mechanic_id - integer

owners
  id - integer
  name - string
  car_id - integer
```

Bây giờ chúng ta đã kiểm tra cấu trúc bảng cho mối quan hệ, hãy xác định mối quan hệ trên model `Mechanic`:

```
<?php

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

class Mechanic extends Model
{
  /**
   * Get the car's owner.
   */
  public function carOwner()
  {
    return $this->hasOneThrough(Owner::class, Car::class);
  }
}
```

Đối số đầu tiên được truyền cho phương thức `hasOneThrough` là tên của model cuối cùng mà chúng ta muốn truy cập, trong khi đối số thứ hai là tên của model trung gian.

Các quy tắc khóa key

Các quy ước khóa ngoại Eloquent điển hình sẽ được sử dụng khi thực hiện các truy vấn của mối quan hệ. Nếu bạn muốn tùy chỉnh các khóa của mối quan hệ, bạn có thể truyền chúng làm đối số thứ ba và thứ tư cho phương thức `hasOneThrough`. Đối số thứ ba là tên của khóa ngoại trên model trung gian. Đối số thứ tư là tên của khóa ngoại trên mô hình cuối cùng. Đối số thứ năm là khóa cục bộ, trong khi đối số thứ sáu là khóa cục bộ của mô hình trung gian:

```
class Mechanic extends Model
{
  /**
   * Get the car's owner.
   */
  public function carOwner()
```

```
  {
    return $this->hasOneThrough(
      Owner::class,
      Car::class,
      'mechanic_id', // Foreign key on the cars table...
      'car_id', // Foreign key on the owners table...
      'id', // Local key on the mechanics table...
      'id' // Local key on the cars table...
    );
  }
}
```

Một qua nhiều

Mối quan hệ "Một qua nhiều" cung cấp một phương pháp thuận tiện để truy cập các quan hệ xa thông qua một quan hệ trung gian. Ví dụ: giả sử chúng tôi đang xây dựng một nền tảng triển khai như Laravel Vapor. Một model `Project` có thể truy cập nhiều model `Deployment` thông qua một model `Enviroment` trung gian. Khi sử dụng ví dụ này, bạn có thể dễ dàng thu thập tất cả các lần triển khai cho một dự án nào đó. Hãy xem cấu trúc các bảng cần thiết để tạo dựng mối quan hệ này:

```
projects
  id - integer
  name - string

environments
  id - integer
  project_id - integer
  name - string

deployments
  id - integer
  environment_id - integer
  commit_hash - string
```

Bây giờ chúng ta đã kiểm tra cấu trúc bảng cho mối quan hệ, hãy xác định mối quan hệ trên model `Project`:

```php
<?php

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

class Project extends Model
{
  /**
   * Get all of the deployments for the project.
   */
  public function deployments()
  {
    return $this->hasManyThrough(Deployment::class, Environment::class);
  }
}
```

Đối số đầu tiên được truyền cho phương thức `hasManyThrough` là tên của model cuối cùng mà chúng ta muốn truy cập, trong khi đối số thứ hai là tên của model trung gian.

Mặc dù bảng của model `Deployment` không chứa cột `project_id`, quan hệ `hasManyThrough` cung cấp quyền truy cập vào các lần triển khai của dự án thông qua `$project->deployments`. Để truy xuất các model này, Eloquent kiểm tra cột `project_id` trên bảng của model `Environment` trung gian. Sau khi tìm thấy các *ID* của môi trường có liên quan, chúng được sử dụng để truy vấn bảng của model `Deployment`.

Các quy tắc khóa key

Các quy ước khóa ngoại Eloquent điển hình sẽ được sử dụng khi thực hiện các truy vấn của mối quan hệ. Nếu bạn muốn tự chọn các khóa của mối quan hệ, bạn có thể truyền chúng làm đối số thứ ba và thứ tư cho phương thức `hasManyThrough`. Đối số thứ ba là tên của khóa ngoại trên model trung gian. Đối số thứ tư là tên của khóa ngoại trên model cuối cùng. Đối số thứ năm là khóa cục bộ, trong khi đối số thứ sáu là khóa cục bộ của model trung gian:

```php
class Project extends Model
{
  public function deployments()
```

```
    {
        return $this->hasManyThrough(
            Deployment::class,
            Environment::class,
            'project_id', // Foreign key on the environments table...
            'environment_id', // Foreign key on the deployments table...
            'id', // Local key on the projects table...
            'id' // Local key on the environments table...
        );
    }
}
```

Mối quan hệ nhiều đối nhiều

Quan hệ nhiều đối nhiều phức tạp hơn một chút so với quan hệ `hasOne` và `hasMany`. Một ví dụ về mối quan hệ nhiều đối nhiều là người dùng có nhiều vai trò và những vai trò đó cũng được chia sẻ bởi những người dùng khác trong ứng dụng. Ví dụ, một người dùng có thể được chỉ định vai trò của "Author" và "Editor"; tuy nhiên, những vai trò đó cũng có thể được chỉ định cho những người dùng khác. Vì vậy, một người dùng có nhiều vai trò và một vai trò có nhiều người dùng.

Cấu trúc bảng

Để xác định mối quan hệ này, cần có ba bảng cơ sở dữ liệu: `users`, `roles` và `role_user`. Bảng `role_user` có nguồn gốc từ thứ tự bảng chữ cái của tên model có liên quan và chứa các cột `user_id` và `role_id`. Bảng này được sử dụng như một bảng trung gian liên kết `users` và `roles`.

Hãy nhớ rằng, vì một vai trò có thể dùng cho nhiều người dùng, chúng ta không thể chỉ đặt cột `user_id` trên bảng vai trò. Điều này có nghĩa là một vai trò chỉ có thể thuộc về một người dùng duy nhất. Để cung cấp hỗ trợ cho các vai trò được gán cho nhiều người dùng, bảng `role_user` là yếu tố cần thiết. Chúng ta có thể tóm tắt cấu trúc bảng của mối quan hệ như sau:

```
users
    id - integer
    name - string
```

```
roles
  id - integer
  name - string

role_user
  user_id - integer
  role_id - integer
```

Model structure

Mối quan hệ nhiều đối nhiều được tạo trên model bằng cách viết một phương thức trả về kết quả của phương thức `belongsToMany`. Phương thức `belongsToMany` được cung cấp bởi lớp cơ sở `Illuminate\Database\Eloquent\Model` được sử dụng bởi tất cả các mô hình Eloquent của ứng dụng của bạn. Ví dụ: hãy tạo phương thức `roles` trên model User của chúng ta. Đối số đầu tiên được truyền cho phương thức này là tên của class model liên quan:

```php
<?php

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

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

Khi mối quan hệ được tạo ra, bạn có thể truy cập vai trò của người dùng bằng cách sử dụng mối quan hệ qua thuộc tính động `roles`:

```php
use App\Models\User;

$user = User::find(1);

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

Vì tất cả các mối quan hệ cũng đóng vai trò là query builder, bạn có thể thêm các ràng buộc khác vào truy vấn mối quan hệ bằng cách gọi phương thức `roles` và tiếp tục xâu chuỗi các vế điều kiện vào truy vấn:

```php
$roles = User::find(1)->roles()->orderBy('name')->get();
```

Để xác định tên của bảng trung gian cho mối quan hệ nhiều đối nhiều, Eloquent sẽ nối hai tên model có liên quan theo thứ tự bảng chữ cái. Tuy nhiên, bạn có thể tự do ghi đè quy ước này. Bạn có thể làm như vậy bằng cách truyền một đối số thứ hai cho phương thức `belongsToMany`:

```php
return $this->belongsToMany(Role::class, 'role_user');
```

Ngoài việc tùy chỉnh tên của bảng trung gian, bạn cũng có thể tùy chỉnh tên cột của các khóa trên bảng bằng cách chuyển các đối số bổ sung cho phương thức `belongsToMany`. Đối số thứ ba là tên khóa ngoại của model mà bạn đang tạo mối quan hệ, trong khi đối số thứ tư là tên khóa ngoại của model mà bạn đang muốn liên kết:

```php
return $this->belongsToMany(Role::class, 'role_user', 'user_id', 'role_id');
```

Phần nghịch đảo của mối quan hệ

Để xác định phần "nghịch đảo" của một mối quan hệ nhiều đối nhiều, bạn nên tạo một phương thức trên model liên quan. Phương thức này cũng trả về kết quả của phương thức `belongsToMany`. Để hoàn thành ví dụ về user/role của chúng ta, hãy xác định phương thức `users` trên model `Role`:

```php
<?php

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

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

Như bạn có thể thấy, mối quan hệ được tạo giống hệt như model đối tác `User` ngoại trừ việc tham chiếu đến model `App\Models\User`. Vì chúng ta đang sử dụng lại phương thức `belongsToMany`, nên tất cả các tùy chọn tùy chỉnh bảng và khóa đều có sẵn khi tạo phần "nghịch đảo" của các mối quan hệ nhiều đối nhiều.

Truy cập vào cột của bảng trung gian

Như bạn đã học, làm việc với quan hệ nhiều đối nhiều yêu cầu phải có một bảng dữ liệu trung gian. Eloquent cung cấp một số cách tương tác rất hữu ích với bảng này. Ví dụ: giả sử model `User` của chúng ta có nhiều model `Role` liên quan đến nó. Sau khi truy cập mối quan hệ này, chúng ta có thể truy cập bảng trung gian bằng cách sử dụng thuộc tính `pivot` trên các model:

```php
use App\Models\User;

$user = User::find(1);

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

Lưu ý rằng mỗi model `Role` mà chúng tôi truy xuất được tự động gán một thuộc tính attribute `pivot`. Thuộc tính atrribute này chứa một model đại diện cho bảng dữ liệu trung gian.

Theo mặc định, chỉ có các phím model mới có trên model `pivot`. Nếu bảng trung gian của bạn chứa các thuộc tính attribute bổ sung, bạn phải chỉ định chúng khi xác định mối quan hệ:

```
return $this->belongsToMany(Role::class)->withPivot('active', 'created_by');
```

Nếu bạn muốn bảng dữ liệu trung gian của mình có dấu thời gian `created_at` và `updated_at` được Eloquent duy trì tự động, hãy gọi phương thức `withTimestamps` khi xác định mối quan hệ:

```
return $this->belongsToMany(Role::class)->withTimestamps();
```

> **Chú ý:** Các bảng trung gian mà sử dụng dấu thời gian được duy trì tự động của Eloquent bắt buộc phải có cả cột thời gian `created_at` và `updated_at`.

Tùy biến tên thuộc tính `pivot`

Như đã lưu ý trước đây, các thuộc tính từ bảng trung gian có thể được truy cập trên các model thông qua thuộc tính attribute `pivot`. Tuy nhiên, bạn có thể tự do tùy chỉnh tên của thuộc tính attribute này để phản ánh tốt hơn mục đích của nó trong ứng dụng của bạn.

Ví dụ: nếu ứng dụng của bạn chứa những người dùng có thể đăng ký podcast, bạn có thể có mối quan hệ nhiều đối nhiều giữa người dùng và podcast. Nếu đúng như vậy, bạn có thể muốn đổi tên thuộc tính attribute bảng trung gian của mình thành `subscription` thay vì `pivot`. Điều này có thể được thực hiện bằng cách sử dụng phương thức `as` khi xác định mối quan hệ:

```
return $this->belongsToMany(Podcast::class)
  ->as('subscription')
  ->withTimestamps();
```

Khi thuộc tính attribute bảng trung gian tự tạo đã được chỉ định, bạn có thể truy cập dữ liệu bảng trung gian bằng tên tùy chỉnh:

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

foreach ($users->flatMap->podcasts as $podcast) {
  echo $podcast->subscription->created_at;
}
```

Lọc các truy vấn qua các cột bảng trung gian

Bạn cũng có thể lọc các kết quả được trả về bởi các truy vấn mối quan hệ `belongsToMany` bằng cách sử dụng các phương thức `wherePivot`, `wherePivotIn`, `wherePivotNotIn`, `wherePivotBetween`, `wherePivotNotBetween`, `wherePivotNull` và `wherePivotNotNull` khi xác định mối quan hệ:

```php
return $this->belongsToMany(Role::class)
                ->wherePivot('approved', 1);

return $this->belongsToMany(Role::class)
                ->wherePivotIn('priority', [1, 2]);

return $this->belongsToMany(Role::class)
                ->wherePivotNotIn('priority', [1, 2]);

return $this->belongsToMany(Podcast::class)
                ->as('subscriptions')
                ->wherePivotBetween('created_at', ['2020-01-01 00:00:00', '2020-12-31 00:00:00']);

return $this->belongsToMany(Podcast::class)
                ->as('subscriptions')
                ->wherePivotNotBetween('created_at', ['2020-01-01 00:00:00', '2020-12-31 00:00:00']);

return $this->belongsToMany(Podcast::class)
                ->as('subscriptions')
                ->wherePivotNull('expired_at');

return $this->belongsToMany(Podcast::class)
                ->as('subscriptions')
```

```
                ->wherePivotNotNull('expired_at');
```

Tự tạo bảng trung gian

Nếu bạn muốn xác định một model tự tạo để đại diện cho bảng trung gian của mối quan hệ nhiều đối nhiều, bạn có thể gọi phương thức `using` khi xác định mối quan hệ. model pivot tự tạo sẽ cung cấp cho bạn cơ hội xác định các phương pháp bổ sung trên model pivot.

Khi bạn tự tạo, các model pivot trong quan hệ nhiều đối nhiều sẽ mở rộng class `Illuminate\Database\Eloquent\Relations\Pivot` trong khi các mô hình pivot nhiều đối nhiều nhưng có tính đa hình sẽ mở rộng class `Illuminate\Database\Eloquent\Relations\MorphPivot`. Ví dụ: chúng ta có thể chỉ định model `Role` sử dụng model trung gian mà bạn tự tạo `RoleUser`:

```php
<?php

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

class Role extends Model
{
  /**
   * The users that belong to the role.
   */
  public function users()
  {
    return $this->belongsToMany(User::class)->using(RoleUser::class);
  }
}
```

Khi xác định mô hình `RoleUser`, bạn nên mở rộng class `Illuminate\Database\Eloquent\Relations\Pivot`:

```php
<?php

namespace App\Models;
use Illuminate\Database\Eloquent\Relations\Pivot;
```

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

> **Chú ý:** Model Pivot có thể không sử dụng trait `SoftDeletes`. Nếu bạn cần xóa mềm các record pivot, hãy xem xét chuyển đổi model pivot của bạn thành một mô hình Eloquent thực tế.

Tùy chỉnh các model pivot và các ID tăng dần

Nếu bạn đã xác định mối quan hệ nhiều đối nhiều mà sử dụng model pivot tự tạo và model pivot đó có khóa chính tự động tăng, bạn nên đảm bảo class model pivot tự tạo của mình xác định thuộc tính `incrementing` được đặt thành `true`.

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

Các mối quan hệ đa hình

Mối quan hệ đa hình cho phép model con có thể được sở hữu bởi nhiều loại model bằng cách sử dụng một liên kết duy nhất. Ví dụ: hãy tưởng tượng bạn đang xây dựng một ứng dụng cho phép người dùng chia sẻ các bài đăng trên blog và video. Trong một ứng dụng như vậy, model Comment có thể thuộc về cả model Post và Video.

Đa hình trong mối quan hệ một đối một

Cấu trúc bảng

Quan hệ đa hình một đối một tương tự như quan hệ một đối một điển hình; tuy nhiên, model con có thể được sở hữu bởi nhiều loại model bằng cách sử dụng một liên kết duy

nhất. Ví dụ: một `Post` trên blog và `User` có thể cùng có quan hệ với model `Photo`. Sử dụng quan hệ đa hình 1-1 cho phép bạn có một bảng images thống nhất có thể được liên kết với các bài đăng và người dùng. Đầu tiên, hãy kiểm tra cấu trúc bảng:

```
posts
  id - integer
  name - string

users
  id - integer
  name - string

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

Lưu ý các cột `imageable_id` và `imageable_type` trên bảng `images`. Cột `imageable_id` sẽ chứa giá trị ID của bài đăng hoặc người dùng, trong khi cột `imageable_type` sẽ chứa tên class của model mẹ. Cột `imageable_type` được sử dụng bởi Eloquent để xác định "kiểu" model mẹ nào sẽ trả về khi truy cập vào quan hệ `imageable`. Trong trường hợp này, cột sẽ chứa `App\Models\Post` hoặc `App\Models\User`.

Cấu trúc model

Tiếp theo, hãy xem xét cách tạo các model cần thiết để xây dựng mối quan hệ này:

```php
<?php

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

class Image extends Model
{
  /**
```

```
     * Get the parent imageable model (user or post).
     */
    public function imageable()
    {
        return $this->morphTo();
    }
}

class Post extends Model
{
    /**
     * Get the post's image.
     */
    public function image()
    {
        return $this->morphOne(Image::class, 'imageable');
    }
}

class User extends Model
{
    /**
     * Get the user's image.
     */
    public function image()
    {
        return $this->morphOne(Image::class, 'imageable');
    }
}
```

Nhận lại mối quan hệ

Khi bảng dữ liệu và các model của bạn được tạo xong, thì bạn có thể truy cập các mối quan hệ thông qua các model của mình. Ví dụ: để truy xuất hình ảnh cho một bài đăng, chúng ta có thể truy cập thuộc tính mối quan hệ động `image`:

```
use App\Models\Post;
```

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

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

Bạn có thể truy xuất nguồn gốc của model đa hình bằng cách truy cập vào tên của phương thức thực hiện lệnh gọi đến `morphTo`. Trong trường hợp này, đó là phương thức `imageable` trên model `Image`. Vì vậy, chúng ta sẽ truy cập phương thức đó dưới dạng thuộc tính quan hệ động:

```php
use App\Models\Image;

$image = Image::find(1);

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

Mối quan hệ imageable trên model Image sẽ trả về đối tượng Post hoặc User, tùy thuộc vào kiểu mô hình nào sở hữu hình ảnh.

Các quy tắc cho khóa liên kết

Nếu cần, bạn có thể chỉ định tên của các cột "id" và "type" được sử dụng bởi model con đa hình. Nếu bạn làm như vậy, hãy đảm bảo rằng bạn luôn truyền tên của mối quan hệ làm đối số đầu tiên cho phương thức `morphTo`. Thông thường, giá trị này phải khớp với tên phương thức, vì vậy bạn có thể sử dụng hằng số `__FUNCTION__` của PHP:

```php
/**
 * Get the model that the image belongs to.
 */
public function imageable()
{
  return $this->morphTo(__FUNCTION__, 'imageable_type', 'imageable_id');
}
```

Quan hệ đa hình một đối nhiều

Cấu trúc bảng

Quan hệ đa hình một đối nhiều tương tự như quan hệ một đối nhiều điển hình; tuy nhiên, model con có thể được sở hữu bởi nhiều loại model bằng cách sử dụng một liên kết duy nhất. Ví dụ: hãy tưởng tượng người dùng ứng dụng của bạn có thể "bình luận" về các bài đăng và video. Sử dụng mối quan hệ đa hình, bạn có thể sử dụng một bảng `comments` thống nhất để chứa nhận xét cho cả bài đăng và video. Đầu tiên, hãy kiểm tra cấu trúc bảng cần thiết để xây dựng mối quan hệ này:

```
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
```

Cấu trúc model

Tiếp theo, hãy xem xét cách tạo các mô hình cần thiết để xây dựng mối quan hệ này:

```php
<?php

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

class Comment extends Model
{
  /**
```

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

class Post extends Model
{
    /**
     * Get all of the post's comments.
     */
    public function comments()
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}

class Video extends Model
{
    /**
     * Get all of the video's comments.
     */
    public function comments()
    {
        return $this->morphMany(Comment::class, 'commentable');
    }
}
```

Truy xuất mối quan hệ

Khi bảng cơ sở dữ liệu và các model của bạn được tạo, bạn có thể truy cập các mối quan hệ thông qua các thuộc tính mối quan hệ động của model của bạn. Ví dụ: để truy cập tất cả các nhận xét cho một bài đăng, chúng ta có thể sử dụng thuộc tính động `comments`:

```
use App\Models\Post;
```

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

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

Bạn cũng có thể truy xuất nguồn gốc của một model đa hình con bằng cách truy cập vào tên của phương thức thực hiện lệnh gọi đến `morphTo`. Trong trường hợp này, đó là phương thức `commentable` trên model `Comment`. Vì vậy, chúng ta sẽ truy cập phương thức đó dưới dạng thuộc tính quan hệ động để truy cập model mẹ của nhận xét:

```php
use App\Models\Comment;

$comment = Comment::find(1);

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

Mối quan hệ `commentable` trên model `Comment` sẽ trả về đối tượng `Post` hoặc `Video`, tùy thuộc vào loại model nào mà nhận xét thuộc về.

Một của nhiều (Đa hình)

Đôi khi một model có thể có nhiều model liên quan, nhưng bạn muốn dễ dàng truy xuất model liên quan "mới nhất" hoặc "cũ nhất" của mối quan hệ. Ví dụ: model User có thể liên quan đến nhiều model Image, nhưng bạn muốn xác định một phương pháp thuận tiện để tương tác với hình ảnh gần đây nhất mà người dùng đã tải lên. Bạn có thể thực hiện điều này bằng cách sử dụng kiểu quan hệ `morphOne` kết hợp với các phương thức `ofMany`:

```php
/**
 * Get the user's most recent image.
 */
public function latestImage()
{
  return $this->morphOne(Image::class, 'imageable')->latestOfMany();
}
```

Tương tự như vậy, bạn có thể xác định một phương pháp để truy xuất model liên quan "cũ nhất" hoặc đầu tiên của mối quan hệ:

```
/**
 * Get the user's oldest image.
 */
public function oldestImage()
{
  return $this->morphOne(Image::class, 'imageable')->oldestOfMany();
}
```

Theo mặc định, các phương thức `latestOfMany` mới nhất và `oldestOfMany` cũ nhất sẽ truy xuất model có liên quan mới nhất hoặc cũ nhất dựa trên khóa chính của model, khóa này phải có thể sắp xếp được. Tuy nhiên, đôi khi bạn có thể muốn truy xuất một model từ một mối quan hệ lớn hơn bằng cách sử dụng một tiêu chí sắp xếp khác.

Ví dụ: khi sử dụng phương thức `ofMany`, bạn có thể truy xuất hình ảnh được "thích" nhất của người dùng. Phương thức `ofMany` chấp nhận cột có thể sắp xếp làm đối số đầu tiên của nó và hàm tổng hợp nào (tối thiểu hoặc tối đa) sẽ áp dụng khi truy vấn model liên quan:

```
/**
 * Get the user's most popular image.
 */
public function bestImage()
{
  return $this->morphOne(Image::class, 'imageable')->ofMany('likes', 'max');
}
```

> Có thể xây dựng các mối quan hệ "một trong nhiều" nâng cao hơn. Để biết thêm thông tin, vui lòng tham khảo phần quan hệ *có một trong nhiều* phía trên.

Quan hệ đa hình nhiều đối nhiều

Cấu trúc bảng

Quan hệ đa hình nhiều đối nhiều phức tạp hơn một chút so với quan hệ "đa hình một" và

"đa hình nhiều". Ví dụ: model `Post` và model `Video` có thể chia sẽ mối quan hệ đa hình với model `Tag`. Việc sử dụng quan hệ đa hình nhiều đối nhiều trong trường hợp này sẽ cho phép ứng dụng của bạn có một bảng các thẻ duy nhất có thể được liên kết với các bài đăng hoặc video. Đầu tiên, hãy kiểm tra cấu trúc bảng cần thiết để xây dựng mối quan hệ này:

```
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
```

> Trước khi đi sâu vào các mối quan hệ đa hình nhiều đối nhiều, bạn có thể được hưởng lợi từ việc đọc tài liệu về các mối quan hệ nhiều đối nhiều điển hình.

Cấu trúc model

Tiếp theo, chúng ta đã sẵn sàng để xác định các mối quan hệ trên các model. Cả hai model `Post` và `Video` sẽ chứa một phương thức `tags` đều gọi đến phương thức `morphToMany` được cung cấp bởi class model Eloquent cơ sở.

Phương thức `morphToMany` chấp nhận tên của model liên quan cũng như "relationship name". Dựa trên tên mà chúng ta đã gán cho tên bảng trung gian của mình và các khóa mà nó chứa, chúng tôi sẽ gọi mối quan hệ là "taggable":

```
<?php
```

```php
namespace App\Models;
use Illuminate\Database\Eloquent\Model;

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

Tạo phần nghịch đảo của mối quan hệ

Tiếp theo, trên `Tag` model, bạn nên xác định một phương thức cho từng model mẹ có thể có của nó. Vì vậy, trong ví dụ này, chúng ta sẽ xác định phương thức `posts` và phương thức `videos`. Cả hai phương thức này sẽ trả về kết quả của phương thức `morphedByMany`.

Phương thức `morphedByMany` chấp nhận tên của model liên quan cũng như "relationship name". Dựa trên tên mà chúng ta đã gán cho tên bảng trung gian của mình và các khóa mà nó chứa, chúng ta sẽ gọi mối quan hệ là "taggable":

```php
<?php

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

class Tag extends Model
{
  /**
   * Get all of the posts that are assigned this tag.
   */
  public function posts()
  {
    return $this->morphedByMany(Post::class, 'taggable');
```

```
    }

    /**
     * Get all of the videos that are assigned this tag.
     */
    public function videos()
    {
        return $this->morphedByMany(Video::class, 'taggable');
    }
}
```

Truy xuất mối quan hệ

Khi bảng cơ sở dữ liệu và các model của bạn được xác định, bạn có thể truy cập các mối quan hệ thông qua các model của mình. Ví dụ: để truy cập tất cả các thẻ cho một bài đăng, bạn có thể sử dụng thuộc tính quan hệ động `tags`:

```
use App\Models\Post;

$post = Post::find(1);

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

Bạn có thể truy xuất cấp độ gốc của một quan hệ đa hình từ model đa hình con bằng cách truy cập vào tên của phương thức thực hiện lệnh gọi đến `morphedByMany`. Trong trường hợp này, đó là các phương thức `posts` hoặc `videos` trên mô hình `Tag`:

```
use App\Models\Tag;

$tag = Tag::find(1);

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

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

Các kiểu đa hình tự tạo

Theo mặc định, Laravel sẽ sử dụng tên class tiêu chuẩn để lưu trữ "loại" của model liên quan. Ví dụ: về mối quan hệ một đối nhiều ở trên trong đó model `Comment` có thể thuộc về model `Post` hoặc `Video`, thì `commentable_type` mặc định sẽ tương ứng là `App\Models\Post` hoặc `App\Models\Video`. Tuy nhiên, bạn cũng có thể muốn tách các giá trị này khỏi cấu trúc bên trong ứng dụng của mình.

Ví dụ: thay vì sử dụng tên model làm "type", chúng ta có thể sử dụng các chuỗi đơn giản như `post` và `video`. Bằng cách làm như vậy, các giá trị cột "type" đa hình trong cơ sở dữ liệu của chúng tôi sẽ vẫn hợp lệ ngay cả khi các model được đổi tên:

```php
use Illuminate\Database\Eloquent\Relations\Relation;

Relation::enforceMorphMap([
  'post' => 'App\Models\Post',
  'video' => 'App\Models\Video',
]);
```

Bạn có thể gọi phương thức `enforceMorphMap` trong phương thức `boot` của class `App\Providers\AppServiceProvider` hoặc tạo một service provider riêng nếu muốn.

Bạn có thể xác định bí danh cho một model đa hình nào đó trong thời gian chạy (runtime) bằng cách sử dụng phương thức `getMorphClass` của mô hình. Ngược lại, bạn có thể xác định tên class đủ điều kiện được liên kết với bí danh morph bằng cách sử dụng phương thức `Relation::getMorphedModel`:

```php
use Illuminate\Database\Eloquent\Relations\Relation;

$alias = $post->getMorphClass();

$class = Relation::getMorphedModel($alias);
```

Chú ý: Khi thêm "bản đồ morph" vào ứng dụng hiện có của bạn, mọi giá trị cột `*_type` có thể pha trộn trong cơ sở dữ liệu của bạn vẫn chứa một class đủ điều kiện sẽ cần được chuyển đổi thành tên "bản đồ" của nó.

Các mối quan hệ động

Bạn có thể sử dụng phương thức `resolveRelationUsing` để xác định mối quan hệ giữa các mô hình Eloquent trong thời gian chạy. Mặc dù thường không được khuyến nghị để phát triển ứng dụng thông thường, nhưng điều này đôi khi có thể hữu ích khi phát triển các gói Laravel.

Phương thức `resolveRelationUsing` chấp nhận tên quan hệ đang cần làm đối số đầu tiên của nó. Đối số thứ hai được truyền cho phương thức phải là một hàm xử lý chấp nhận đối tượng model và trả về một kết quả quan hệ Eloquent hợp lệ. Thông thường, bạn nên cấu hình các mối quan hệ động trong phương thức `boot` của service provider:

```
use App\Models\Order;
use App\Models\Customer;

Order::resolveRelationUsing('customer', function ($orderModel) {
  return $orderModel->belongsTo(Customer::class, 'customer_id');
});
```

Chú ý: Khi xác định mối quan hệ động, hãy luôn cung cấp các đối số tên khóa rõ ràng cho các phương thức quan hệ Eloquent.

Truy vấn các quan hệ

Vì tất cả các mối quan hệ Eloquent được xác định thông qua các phương thức, nên bạn có thể gọi các phương thức đó để lấy một đối tượng của mối quan hệ mà không thực sự thực hiện một truy vấn để tải các model liên quan. Ngoài ra, tất cả các loại mối quan hệ Eloquent cũng đóng vai trò là trình tạo truy vấn, cho phép bạn tiếp tục xâu chuỗi các ràng buộc đối với truy vấn mối quan hệ trước khi thực hiện truy vấn SQL đối với cơ sở dữ liệu của bạn.

Ví dụ: hãy tưởng tượng một ứng dụng blog trong đó `User` model có nhiều model `Post` được liên kết:

```php
<?php

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

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

Bạn có thể truy vấn mối quan hệ `posts` và thêm các ràng buộc bổ sung vào mối quan hệ như sau:

```php
use App\Models\User;

$user = User::find(1);

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

Bạn có thể sử dụng bất kỳ phương thức nào của trình tạo truy vấn Laravel trên mối quan hệ, vì vậy hãy đảm bảo khám phá tài liệu về trình tạo truy vấn để tìm hiểu về tất cả các phương thức có sẵn cho bạn.

Xâu chuỗi mệnh đề `orWhere` vào sau mối quan hệ

Như đã trình bày trong ví dụ trên, bạn có thể tự do thêm các ràng buộc bổ sung vào các mối quan hệ khi truy vấn chúng. Tuy nhiên, hãy thận trọng khi xâu chuỗi các mệnh đề `orWhere` vào một mối quan hệ, vì các mệnh đề `orWhere` sẽ được nhóm một cách hợp lý ở cùng cấp với ràng buộc mối quan hệ:

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

Ví dụ trên sẽ tạo ra SQL sau. Như bạn có thể thấy, mệnh đề `or` hướng dẫn truy vấn trả về bất kỳ người dùng nào có hơn 100 phiếu bầu. Truy vấn không còn bị ràng buộc đối với một người dùng cụ thể:

```sql
select *
from posts
where user_id = ? and active = 1 or votes >= 100
```

Trong hầu hết các tình huống, bạn nên sử dụng các nhóm logic để nhóm các kiểm tra có điều kiện giữa các dấu ngoặc đơn:

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

$user->posts()
  ->where(function (Builder $query) {
    return $query->where('active', 1)
      ->orWhere('votes', '>=', 100);
  })
  ->get();
```

Ví dụ trên sẽ tạo ra SQL sau. Lưu ý rằng nhóm logic đã nhóm các ràng buộc đúng cách và truy vấn vẫn bị hạn chế đối với một người dùng cụ thể:

```sql
select *
from posts
where user_id = ? and (active = 1 or votes >= 100)
```

Các phương thức mối quan hệ và các thuộc tính động

Nếu bạn không cần thêm các ràng buộc bổ sung vào truy vấn mối quan hệ Eloquent, bạn có

thể truy cập mối quan hệ như thể nó là một thuộc tính. Ví dụ: tiếp tục sử dụng các model ví dụ về `User` và `Post` của chúng ta, chúng ta có thể truy cập tất cả các bài đăng của người dùng như vậy:

```
use App\Models\User;

$user = User::find(1);

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

Thuộc tính mối quan hệ động thực hiện "tải chậm", có nghĩa là chúng sẽ chỉ tải dữ liệu mối quan hệ khi bạn thực sự truy cập chúng. Bởi vì điều này, các nhà phát triển thường sử dụng tải eager loading để tải trước các mối quan hệ mà họ biết sẽ được truy cập sau khi tải mô hình. Tải eager loading giúp giảm đáng kể các truy vấn SQL phải được thực thi để tải các quan hệ của model.

Truy vấn sự tồn tại của mối quan hệ

Khi truy xuất các record của model, bạn có thể muốn giới hạn kết quả của mình dựa trên sự tồn tại của một mối quan hệ. Ví dụ: hãy tưởng tượng bạn muốn truy xuất tất cả các bài đăng trên blog có ít nhất một bình luận. Để làm như vậy, bạn có thể truyền tên của mối quan hệ vào phương thức `doesntHave` và `orDoesntHave`:

```
use App\Models\Post;

// Retrieve all posts that have at least one comment...
$posts = Post::has('comments')->get();
```

Bạn cũng có thể chỉ định một toán tử và giá trị đếm để tùy chỉnh thêm truy vấn:

```
// Retrieve all posts that have three or more comments...
$posts = Post::has('comments', '>=', 3)->get();
```

Các câu lệnh có lồng nhau có thể được xây dựng bằng cách sử dụng ký hiệu "dấu chấm". Ví

dụ: bạn có thể truy xuất tất cả các bài đăng có ít nhất một nhận xét có ít nhất một hình ảnh:

```
// Retrieve posts that have at least one comment with images...
$posts = Post::has('comments.images')->get();
```

Nếu bạn cần thêm sức mạnh, bạn có thể sử dụng các phương thức `whereHas` và `orWhereHas` để xác định các ràng buộc truy vấn bổ sung trên các truy vấn has của bạn, chẳng hạn như kiểm tra nội dung của một nhận xét:

```
use Illuminate\Database\Eloquent\Builder;

// Retrieve posts with at least one comment containing words like code%...
$posts = Post::whereHas('comments', function (Builder $query) {
  $query->where('content', 'like', 'code%');
})->get();

// Retrieve posts with at least ten comments containing words like code%...
$posts = Post::whereHas('comments', function (Builder $query) {
  $query->where('content', 'like', 'code%');
}, '>=', 10)->get();
```

> **Chú ý:** Eloquent hiện không hỗ trợ truy vấn về sự tồn tại của mối quan hệ trên các cơ sở dữ liệu. Các mối quan hệ phải tồn tại trong cùng một cơ sở dữ liệu.

Truy vấn nhanh sự tồn tại mối quan hệ

Nếu bạn muốn truy vấn sự tồn tại của một mối quan hệ với một điều kiện where đơn giản được đính kèm với truy vấn mối quan hệ, bạn có thể thấy thuận tiện hơn khi sử dụng các phương thức `whereRelation` và `whereMorphRelation`. Ví dụ: chúng tôi có thể truy vấn tất cả các bài đăng có nhận xét chưa được duyệt:

```
use App\Models\Post;

$posts = Post::whereRelation('comments', 'is_approved', false)->get();
```

Tất nhiên, giống như các lệnh gọi đến phương thức `where` của trình tạo truy vấn, bạn cũng

có thể chỉ định một toán tử:

```
$posts = Post::whereRelation(
  'comments', 'created_at', '>=', now()->subHour()
)->get();
```

Truy vấn sự thiếu mặt của mối quan hệ

Khi truy xuất record với model, bạn có thể muốn giới hạn kết quả của mình dựa trên sự vắng mặt của mối quan hệ. Ví dụ, hãy tưởng tượng bạn muốn truy xuất tất cả các bài đăng trên blog mà không có bất kỳ nhận xét nào. Để làm như vậy, bạn có thể truyền tên của mối quan hệ vào các phương thức `doesntHave` và `orDoesntHave`:

```
use App\Models\Post;

$posts = Post::doesntHave('comments')->get();
```

Nếu bạn cần nhiều sức mạnh hơn nữa, bạn có thể sử dụng các phương thức `whereDoesntHave` và `orWhereDoesntHave` để thêm các ràng buộc truy vấn bổ sung vào các truy vấn `doesntHave` của mình, chẳng hạn như kiểm tra nội dung của một nhận xét:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::whereDoesntHave('comments', function (Builder $query) {
  $query->where('content', 'like', 'code%');
})->get();
```

Bạn có thể sử dụng ký hiệu "dấu chấm" để thực hiện truy vấn chống lại mối quan hệ lồng nhau. Ví dụ, truy vấn sau sẽ lấy tất cả các bài đăng không có bình luận; tuy nhiên, những bài viết có nhận xét của các tác giả không bị cấm sẽ được đưa vào kết quả:

```
use Illuminate\Database\Eloquent\Builder;

$posts = Post::whereDoesntHave('comments.author', function (Builder $query) {
  $query->where('banned', 0);
```

```
})->get();
```

Truy vấn mối quan hệ morph to

Để truy vấn sự tồn tại của các mối quan hệ "morph to", bạn có thể sử dụng phương thức `whereHasMorph` và `whereDoesntHaveMorph`. Các phương thức này chấp nhận tên của mối quan hệ làm đối số đầu tiên của chúng. Tiếp theo, các phương thức chấp nhận tên của các model liên quan mà bạn muốn đưa vào truy vấn. Cuối cùng, bạn có thể cung cấp một hàm xử lý để tùy chỉnh truy vấn mối quan hệ:

```
use App\Models\Comment;
use App\Models\Post;
use App\Models\Video;
use Illuminate\Database\Eloquent\Builder;

// Retrieve comments associated to posts or videos with a title like code%...
$comments = Comment::whereHasMorph(
  'commentable',
  [Post::class, Video::class],
  function (Builder $query) {
    $query->where('title', 'like', 'code%');
  }
)->get();

// Retrieve comments associated to posts with a title not like code%...
$comments = Comment::whereDoesntHaveMorph(
  'commentable',
  Post::class,
  function (Builder $query) {
    $query->where('title', 'like', 'code%');
  }
)->get();
```

Đôi khi bạn có thể cần thêm các ràng buộc truy vấn dựa trên "loại" của model đa hình có liên quan. Hàm xử lý được truyền cho phương thức `whereHasMorph` có thể nhận giá trị `$type` làm đối số thứ hai của nó. Đối số này cho phép bạn kiểm tra "loại" của truy vấn đang được tạo:

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

$comments = Comment::whereHasMorph(
  'commentable',
  [Post::class, Video::class],
  function (Builder $query, $type) {
    $column = $type === Post::class ? 'content' : 'title';

    $query->where($column, 'like', 'code%');
  }
)->get();
```

Truy vấn tất cả model liên quan

Thay vì truyền một mảng các model đa hình có thể có, bạn có thể cung cấp * làm giá trị ký tự đại diện. Điều này sẽ hướng dẫn Laravel truy xuất tất cả các kiểu đa hình có thể có từ cơ sở dữ liệu. Laravel sẽ thực hiện một truy vấn bổ sung để thực hiện thao tác này:

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

$comments = Comment::whereHasMorph('commentable', '*', function (Builder $query) {
  $query->where('title', 'like', 'foo%');
})->get();
```

Tổng hợp các model liên quan

Đếm các model liên quan

Đôi khi bạn có thể muốn đếm số lượng các model liên quan cho một mối quan hệ nhất định mà không thực sự tải các model. Để thực hiện điều này, bạn có thể sử dụng phương thức `withCount`. Phương thức `withCount` sẽ đặt thuộc tính attribute {relation}_count trên các model kết quả:

```php
use App\Models\Post;
```

```php
$posts = Post::withCount('comments')->get();

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

Bằng cách truyền một mảng sang phương thức `withCount`, bạn có thể thêm "số lượng" cho nhiều quan hệ cũng như thêm các ràng buộc bổ sung vào các truy vấn:

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

$posts = Post::withCount(['votes', 'comments' => function (Builder $query) {
  $query->where('content', 'like', 'code%');
}])->get();

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

Bạn cũng có thể đặt bí danh cho kết quả đếm mối quan hệ, cho phép nhiều lần đếm trên cùng một mối quan hệ:

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

$posts = Post::withCount([
  'comments',
  'comments as pending_comments_count' => function (Builder $query) {
    $query->where('approved', false);
  },
])->get();

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

Đếm sau khi truy xuất

Sử dụng phương thức `loadCount`, bạn có thể tải số lượng mối quan hệ sau khi model gốc

đã được truy xuất:

```
$book = Book::first();

$book->loadCount('genres');
```

Nếu bạn cần đặt các ràng buộc truy vấn bổ sung trên truy vấn đếm, bạn có thể chuyển một mảng được khóa bởi các mối quan hệ mà bạn muốn đếm. Các giá trị mảng phải là các hàm xử lý mà sẽ nhận đối tượng query builder:

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

Đếm mối quan hệ & tùy chỉnh các phát biểu select

Nếu bạn đang kết hợp `withCount` với một câu lệnh `select`, hãy đảm bảo rằng bạn gọi `withCount` sau phương thức `select`:

```
$posts = Post::select(['title', 'body'])
  ->withCount('comments')
  ->get();
```

Các hàm tổng hợp khác

Ngoài phương thức `withCount`, Eloquent cung cấp các phương thức `withMin`, `withMax`, `withAvg`, `withSum` và `withExists`. Các phương thức này sẽ đặt thuộc tính {relation}_{function} _{column} trên các model kết quả của bạn:

```
use App\Models\Post;

$posts = Post::withSum('comments', 'votes')->get();

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

```
}
```

Nếu bạn muốn truy cập kết quả của hàm tổng hợp bằng tên khác, bạn có thể chỉ định bí danh của riêng mình:

```
$posts = Post::withSum('comments as total_comments', 'votes')->get();

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

Giống như phương thức `loadCount`, các đối tượng trì hoãn của các phương thức này cũng có sẵn. Các hoạt động tổng hợp bổ sung này có thể được thực hiện trên các model Eloquent đã được truy xuất:

```
$post = Post::first();

$post->loadSum('comments', 'votes');
```

Nếu bạn đang kết hợp các phương thức tổng hợp này với một câu lệnh `select`, hãy đảm bảo rằng bạn gọi các phương thức tổng hợp sau phương thức `select`:

```
$posts = Post::select(['title', 'body'])
  ->withExists('comments')
  ->get();
```

Đếm model liên quan trên các mối quan hệ đa hình

Nếu bạn muốn tải một mối quan hệ "morph to", cũng như số lượng mô hình liên quan cho các thực thể khác nhau có thể được trả về bởi mối quan hệ đó, bạn có thể sử dụng phương thức `with` kết hợp với phương thức `morphWithCount` của mối quan hệ `morphTo`.

Trong ví dụ này, giả sử rằng các mô hình `Photo` và `Post` có thể tạo ra các model `ActivityFeed`. Chúng ta sẽ giả sử model `ActivityFeed` xác định mối quan hệ "morph to" có tên là `parentable` cho phép chúng ta truy xuất model `Photo` hoặc `Post` cho một đối tượng `ActivityFeed` nhất định. Ngoài ra, hãy giả sử rằng model `Photo` "có nhiều"

model `Tag` và model `Post` "có nhiều" model `Comment`.

Bây giờ, hãy tưởng tượng chúng ta muốn truy xuất các cá thể `ActivityFeed` và tải các model mẹ `parentable` cho mỗi cá thể `ActivityFeed`. Ngoài ra, chúng ta muốn truy xuất số lượng thẻ được liên kết với mỗi ảnh gốc và số nhận xét được liên kết với mỗi bài đăng gốc:

```
use Illuminate\Database\Eloquent\Relations\MorphTo;

$activities = ActivityFeed::with([
  'parentable' => function (MorphTo $morphTo) {
    $morphTo->morphWithCount([
      Photo::class => ['tags'],
      Post::class => ['comments'],
    ]);
  }])->get();
```

Đếm sau khi truy vấn

Giả sử chúng ta đã truy xuất một tập hợp các model `ActivityFeed` và bây giờ chúng ta muốn tải số lượng mối quan hệ lồng nhau cho các model `parentable` khác nhau được liên kết với nguồn cấp dữ liệu hoạt động. Bạn có thể sử dụng phương thức `loadMorphCount` để thực hiện điều này:

```
$activities = ActivityFeed::with('parentable')->get();

$activities->loadMorphCount('parentable', [
  Photo::class => ['tags'],
  Post::class => ['comments'],
]);
```

Eager Loading

Khi truy cập các mối quan hệ Eloquent dưới dạng thuộc tính, các model liên quan được "tải chậm". Điều này có nghĩa là dữ liệu mối quan hệ không thực sự được tải cho đến khi bạn truy cập thuộc tính lần đầu tiên. Tuy nhiên, Eloquent vẫn có thể "tải nhanh" các mối quan hệ tại thời điểm bạn truy vấn model mẹ. Tải nhanh làm giảm bớt vấn đề truy vấn "N + 1".

Để minh họa vấn đề truy vấn N + 1, hãy xem xét model Book "thuộc về" Author model:

```php
<?php

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

class Book extends Model
{
  /**
   * Get the author that wrote the book.
   */
  public function author()
  {
    return $this->belongsTo(Author::class);
  }
}
```

Bây giờ, hãy truy xuất tất cả sách và tác giả của chúng:

```php
use App\Models\Book;

$books = Book::all();

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

Vòng lặp này sẽ thực hiện một truy vấn để truy xuất tất cả sách trong bảng cơ sở dữ liệu, sau đó truy xuất một truy vấn khác cho từng sách để truy xuất tác giả của sách. Vì vậy, nếu chúng ta có 25 cuốn sách, đoạn mã trên sẽ chạy 26 truy vấn: một cho cuốn sách gốc và 25 truy vấn bổ sung để truy xuất tác giả của mỗi cuốn sách.

Rất may, chúng ta có thể sử dụng tải nhanh để giảm hoạt động này xuống chỉ còn hai truy vấn. Khi xây dựng một truy vấn, bạn có thể chỉ định những mối quan hệ nào nên được tải bằng cách sử dụng phương thức `with`:

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

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

Đối với thao tác này, chỉ có hai truy vấn sẽ được thực hiện - một truy vấn để truy xuất tất cả các sách và một truy vấn để truy xuất tất cả các tác giả cho tất cả các sách:

```sql
select * from books

select * from authors where id in (1, 2, 3, 4, 5, ...)
```

Tải nhanh nhiều mối quan hệ

Đôi khi bạn có thể cần phải tải một số mối quan hệ khác nhau. Để làm như vậy, chỉ cần truyền một mảng các mối quan hệ tới phương thức `with`:

```php
$books = Book::with(['author', 'publisher'])->get();
```

Lồng tải nhanh

Để tải nhanh các mối quan hệ của một mối quan hệ, bạn có thể sử dụng cú pháp "dấu chấm". Ví dụ: hãy tải nhanh tất cả các tác giả của cuốn sách và tất cả các địa chỉ liên hệ cá nhân của tác giả:

```php
$books = Book::with('author.contacts')->get();
```

Lồng tải nhanh các mối quan hệ `morphTo`

Nếu bạn muốn tải nhanh mối quan hệ đa hình `morphTo`, cũng như các mối quan hệ lồng nhau trên các thực thể khác nhau có thể được trả về bởi mối quan hệ đó, bạn có thể sử dụng phương thức `with` kết hợp với phương thức `morphWith` của mối quan hệ `morphTo`. Để giúp minh họa phương pháp này, chúng ta hãy xem xét model sau:

```php
<?php

use Illuminate\Database\Eloquent\Model;

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

Trong ví dụ này, giả sử các model `Event`, `Photo` và `Post` có thể tạo ra các model `ActivityFeed`. Ngoài ra, giả sử rằng model `Event` thuộc về model `Calendar`, model `Photo` được liên kết với model `Tag` và model `Post` thuộc về model `Author`.

Khi sử dụng các khai báo và mối quan hệ của model này, chúng ta có thể truy xuất các đối tượng model `ActivityFeed` và tải nhanh tất cả các model `parentable` và các mối quan hệ lồng nhau tương ứng của chúng:

```php
use Illuminate\Database\Eloquent\Relations\MorphTo;

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

Tải nhanh các cột cụ thể

Không phải lúc nào bạn cũng cần mọi cột từ các mối quan hệ mà bạn đang truy xuất. Vì lý do này, Eloquent cho phép bạn chỉ đích những cột của mối quan hệ mà bạn muốn truy xuất:

```
$books = Book::with('author:id,name,book_id')->get();
```

> **Chú ý:** Khi sử dụng tính năng này, bạn phải luôn bao gồm cột `id` và bất kỳ cột khóa ngoại nào có liên quan trong danh sách các cột bạn muốn truy xuất.

Cài đặt tải nhanh mặc định

Đôi khi bạn có thể muốn ứng dụng của mình luôn tải nhanh một số mối quan hệ khi truy xuất một model. Để thực hiện điều này, bạn có thể xác định thuộc tính `$with` trên model:

```
<?php

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

class Book extends Model
{
  /**
   * The relationships that should always be loaded.
   *
   * @var array
   */
  protected $with = ['author'];

  /**
   * Get the author that wrote the book.
   */
  public function author()
  {
    return $this->belongsTo(Author::class);
  }

  /**
```

```
     * Get the genre of the book.
     */
    public function genre()
    {
        return $this->belongsTo(Genre::class);
    }
}
```

Nếu bạn muốn xóa bỏ một mục khỏi thuộc tính `$with` cho một truy vấn đơn lẻ, bạn có thể sử dụng phương thức `without`:

```
$books = Book::without('author')->get();
```

Nếu bạn muốn ghi đè tất cả các mục trong thuộc tính `$with` cho một truy vấn đơn lẻ, bạn có thể sử dụng phương thức `withOnly`:

```
$books = Book::withOnly('genre')->get();
```

Ràng buộc tải nhanh

Đôi khi bạn có thể muốn tải nhanh một mối quan hệ nhưng cũng chỉ định các điều kiện truy vấn bổ sung cho truy vấn tải nhanh. Bạn có thể thực hiện điều này bằng cách truyền một mảng các mối quan hệ tới phương thức `with` trong đó khóa mảng là tên mối quan hệ và giá trị mảng là một hàm xử lý để thêm các ràng buộc bổ sung vào truy vấn tải nhanh:

```
use App\Models\User;

$users = User::with(['posts' => function ($query) {
    $query->where('title', 'like', '%code%');
}])->get();
```

Trong ví dụ này, Eloquent sẽ chỉ tải nhanh các bài đăng có cột `title` của bài đăng chứa từ `code`. Bạn có thể gọi các phương thức của query builder khác để tùy chỉnh thêm hoạt động tải nhanh:

```php
$users = User::with(['posts' => function ($query) {
  $query->orderBy('created_at', 'desc');
}])->get();
```

> **Chú ý:** Phương thức của query builder `limit` và `take` có thể không được sử dụng khi ràng buộc việc tải nhanh.

Ràng buộc tải nhanh trong các mối quan hệ `morphTo`

Nếu bạn đang muốn tải một mối quan hệ `morphTo`, Eloquent sẽ chạy nhiều truy vấn để tìm nạp từng loại model có liên quan. Bạn có thể thêm các ràng buộc bổ sung cho mỗi truy vấn này bằng cách sử dụng phương thức `constrain` của quan hệ `MorphTo`:

```php
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Relations\MorphTo;

$comments = Comment::with(['commentable' => function (MorphTo $morphTo) {
  $morphTo->constrain([
    Post::class => function (Builder $query) {
      $query->whereNull('hidden_at');
    },
    Video::class => function (Builder $query) {
      $query->where('type', 'educational');
    },
  ]);
}])->get();
```

Trong ví dụ này, Eloquent sẽ chỉ tải các bài đăng chưa bị ẩn và video có giá trị `type` là "giáo dục".

Tải chậm Eager

Đôi khi bạn có thể cần phải tải một mối quan hệ sau khi model gốc đã được truy xuất. Ví dụ: điều này có thể hữu ích nếu bạn cần tự động quyết định xem có tải các model liên quan hay không:

```php
use App\Models\Book;

$books = Book::all();

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

Nếu bạn cần đặt các ràng buộc truy vấn bổ sung trên truy vấn đang tải mong muốn, bạn có thể truyền một mảng được khóa bởi các mối quan hệ mà bạn muốn tải. Các giá trị mảng phải là các hàm xử lý mà sẽ nhận về đối tượng truy vấn:

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

Để tải một mối quan hệ chỉ khi nó chưa được tải, hãy sử dụng phương thức `loadMissing`:

```php
$book->loadMissing('author');
```

Lồng tải chậm eager và `morphTo`

Nếu bạn muốn tải một mối quan hệ `morphTo`, cũng như các mối quan hệ lồng nhau trên các thực thể khác nhau có thể được trả về bởi mối quan hệ đó, bạn có thể sử dụng phương thức `loadMorph`.

Phương thức này chấp nhận tên của mối quan hệ `morphTo` làm đối số đầu tiên của nó và một mảng các cặp model/*mối quan hệ* làm đối số thứ hai của nó. Để giúp minh họa phương thức này, chúng ta hãy xem xét model sau:

```php
<?php

use Illuminate\Database\Eloquent\Model;

class ActivityFeed extends Model
{
```

```
    /**
     * Get the parent of the activity feed record.
     */
    public function parentable()
    {
        return $this->morphTo();
    }
}
```

Trong ví dụ này, giả sử các model `Event`, `Photo` và `Post` có thể tạo ra các model `ActivityFeed`. Ngoài ra, giả sử rằng model `Event` thuộc về model `Calendar`, model `Photo` được liên kết với model `Tag` và model `Post` thuộc về model `Author`.

Sử dụng các khai báo và mối quan hệ của model này, chúng ta có thể truy xuất các đối tượng model `ActivityFeed` và tải nhanh tất cả các model `parentable` và các mối quan hệ lồng nhau tương ứng của chúng:

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

Ngăn chặn tải chậm

Như đã thảo luận trước đây, các mối quan hệ tải nhanh thường có thể mang lại hiệu suất đáng kể cho ứng dụng của bạn. Do đó, nếu muốn, bạn có thể hướng dẫn Laravel luôn ngăn chặn việc tải chậm các mối quan hệ. Để thực hiện điều này, bạn có thể gọi phương thức `preventLazyLoading` được cung cấp bởi class model Eloquent cơ sở. Thông thường, bạn nên gọi phương thức này trong phương thức boot của class `AppServiceProvider` của ứng dụng.

Phương thức `preventLazyLoading` chấp nhận đối số *boolean* (không bắt buộc) cho biết có nên ngăn chặn tải chậm hay không. Ví dụ: bạn có thể chỉ muốn tắt tính năng tải chậm trong môi trường chưa phát hành (non-production) để môi trường product của bạn tiếp tục hoạt động bình thường ngay cả khi mối quan hệ tải chậm vô tình xuất hiện trong mã thành

phẩm:

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

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
  Model::preventLazyLoading(! $this->app->isProduction());
}
```

Sau khi ngăn chặn tải chậm, Eloquent sẽ đưa ra một ngoại lệ `Illuminate\Database\LazyLoadingViolationException` khi ứng dụng của bạn cố gắng tải chậm bất kỳ mối quan hệ nào của Eloquent.

Bạn có thể tùy chỉnh hành vi của việc tải chậm bằng cách sử dụng phương thức `handleLazyLoadingViolationsUsing`. Ví dụ: bằng cách sử dụng phương pháp này, bạn có thể hướng dẫn việc tải chậm chỉ được ghi lại thay vì làm gián đoạn quá trình thực thi của ứng dụng với các ngoại lệ:

```
Model::handleLazyLoadingViolationUsing(function ($model, $relation) {
  $class = get_class($model);

  info("Attempted to lazy load [{$relation}] on model [{$class}].");
});
```

Chèn và cập nhật các model liên quan

Phương thức `save`

Eloquent cung cấp các phương pháp thuận tiện để thêm các model mới vào các mối quan hệ. Ví dụ: có thể bạn cần thêm bình luận mới vào bài đăng. Thay vì cài đặt thủ công thuộc tính attribute `post_id` trên model `Comment`, bạn có thể chèn thêm bình luận bằng cách sử

dụng phương thức `save` của mối quan hệ:

```
use App\Models\Comment;
use App\Models\Post;

$comment = new Comment(['message' => 'A new comment.']);

$post = Post::find(1);

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

Lưu ý rằng chúng tôi không truy cập mối quan hệ `comments` dưới dạng thuộc tính động. Thay vào đó, chúng tôi gọi phương thức `comments` để nhận một đối tượng của mối quan hệ. Phương thức `save` sẽ tự động thêm giá trị `post_id` thích hợp vào model `Comment` mới.

Nếu bạn cần lưu nhiều model liên quan, bạn có thể sử dụng phương thức `saveMany`:

```
$post = Post::find(1);

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

Các phương thức `save` và `saveMany` sẽ duy trì các đối tượng model đã cho, nhưng sẽ không thêm các model mới được duy trì vào bất kỳ mối quan hệ trong bộ nhớ nào đã được tải vào model mẹ. Nếu bạn định truy cập mối quan hệ sau khi sử dụng phương thức `save` hoặc `saveMany`, bạn có thể muốn sử dụng phương thức `refresh` để tải lại model và các mối quan hệ của nó:

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

$post->refresh();

// All comments, including the newly saved comment...
$post->comments;
```

Đệ quy trong việc lưu model và mối quan hệ

Nếu bạn muốn lưu model của mình và tất cả các mối quan hệ liên quan của nó, bạn có thể sử dụng phương thức `push`. Trong ví dụ này, model `Post` sẽ được lưu cũng như các bình luận của nó và tác giả của bình luận:

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

$post->comments[0]->message = 'Message';
$post->comments[0]->author->name = 'Author Name';

$post->push();
```

Phương thức `create`

Ngoài các phương thức `save` và `saveMany`, bạn cũng có thể sử dụng phương thức `create`, phương thức này chấp nhận một mảng thuộc tính attribute, tạo một model và chèn nó vào cơ sở dữ liệu. Sự khác biệt giữa `save` và `create` đó là `save` chấp nhận một đối tượng model Eloquent đầy đủ trong khi `create` chấp nhận một mảng PHP thuần túy. Model mới được tạo sẽ được trả về bởi phương thức `create`:

```php
use App\Models\Post;

$post = Post::find(1);

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

Bạn có thể sử dụng phương thức `createMany` để tạo nhiều model có liên quan:

```
$post = Post::find(1);

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

Bạn cũng có thể sử dụng các phương thức `findOrNew`, `firstOrNew`, `firstOrCreate` và `updateOrCreate` để tạo và cập nhật các model trên các mối quan hệ.

> Trước khi sử dụng phương thức `create`, hãy nhớ xem lại tài liệu mass assignment.

Các mối quan hệ belongs To

Nếu bạn muốn gán một model con cho một model mẹ mới, bạn có thể sử dụng phương thức `associate`. Trong ví dụ này, model `User` xác định mối quan hệ `belongsTo` với `Account` model. Phương thức `associate` này sẽ đặt khóa ngoại trên model con:

```
use App\Models\Account;

$account = Account::find(10);

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

$user->save();
```

Để loại bỏ model mẹ khỏi model con, bạn có thể sử dụng phương thức `dissociate`. Phương thức này sẽ đặt khóa ngoại của mối quan hệ thành `null`:

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

$user->save();
```

Các mối quan hệ nhiều đối nhiều

Đính kèm / Tách rời

Eloquent còn cung cấp các phương thức giúp làm việc với nhiều mối quan hệ thuận tiện hơn. Ví dụ, hãy tưởng tượng một người dùng có thể có nhiều vai trò và một vai trò có thể có nhiều người dùng. Bạn có thể sử dụng phương thức `attach` để đính kèm vai trò cho người dùng bằng cách insert một record vào bảng dữ liệu trung gian của mối quan hệ:

```
use App\Models\User;

$user = User::find(1);

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

Khi đính kèm mối quan hệ với một model, bạn cũng có thể truyền một mảng dữ liệu bổ sung để chèn vào bảng trung gian:

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

Đôi khi có thể cần phải xóa một vai trò khỏi người dùng. Để xóa record mối quan hệ nhiều đối nhiều, hãy sử dụng phương thức `detach`. Phương thức `detach` sẽ xóa record thích hợp ra khỏi bảng trung gian; tuy nhiên, cả hai model sẽ vẫn còn trong cơ sở dữ liệu:

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

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

Để thuận tiện, `attach` và `detach` cũng chấp nhận các mảng ID làm đầu vào:

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

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

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

Đồng bộ hóa các mối liên kết

Bạn cũng có thể sử dụng phương thức `sync` để tạo liên kết nhiều đối nhiều. Phương thức `sync` chấp nhận một mảng ID để đặt trên bảng trung gian. Bất kỳ ID nào không nằm trong mảng đã cho sẽ bị xóa khỏi bảng trung gian. Vì vậy, sau khi hoạt động này hoàn tất, chỉ các ID trong mảng đã cho sẽ tồn tại trong bảng trung gian:

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

Bạn cũng có thể truyền các giá trị bảng trung gian bổ sung với các ID:

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

Nếu bạn muốn chèn các giá trị bảng trung gian giống nhau với mỗi ID model được đồng bộ hóa, bạn có thể sử dụng phương thức `syncWithPivotValues`:

```php
$user->roles()->syncWithPivotValues([1, 2, 3], ['active' => true]);
```

Nếu bạn không muốn tách các ID hiện có bị thiếu khỏi mảng đã cho, bạn có thể sử dụng phương thức `syncWithoutDetaching`:

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

Chuyển đổi các mối liên kết

Mối quan hệ nhiều đối nhiều cũng cung cấp một phương thức `toggle` để "chuyển đổi" trạng thái đính kèm của các ID model có liên quan đã cho. Nếu ID đã cho hiện đang được đính kèm, nó sẽ bị tách ra. Tương tự như vậy, nếu nó hiện đang được tách ra, nó sẽ được đính kèm:

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

Nâng cấp một record trên bảng dữ liệu trung gian

Nếu bạn cần cập nhật một record hiện có trong bảng dữ liệu trung gian của mối quan hệ của mình, bạn có thể sử dụng phương thức `updateExistingPivot`. Phương thức này chấp nhận khóa ngoại của record trung gian và một mảng thuộc tính attribute để cập nhật:

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

$user->roles()->updateExistingPivot($roleId, [
  'active' => false,
]);
```

Chạm vào Dấu thời gian gốc

Khi một model xác định mối quan hệ `belongsTo` hoặc `belongsToMany` đối với model khác, chẳng hạn như `Comment` thuộc về một bài `Post`, đôi khi sẽ hữu ích nếu cập nhật dấu thời gian của model mẹ khi model con được cập nhật.

Ví dụ: khi model `Comment` được cập nhật, bạn có thể muốn tự động đóng dấu thời gian `updated_at` của bài `Post` sở hữu bình luận đó để nó được đặt thành ngày và giờ hiện tại. Để thực hiện điều này, bạn có thể thêm thuộc tính `touches` vào model con của mình đang chứa tên của các mối quan hệ sẽ được cập nhật dấu thời gian `updated_at` của chúng khi model con được cập nhật:

```
<?php

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

class Comment extends Model
{
  /**
   * All of the relationships to be touched.
   *
```

```php
     * @var array
     */
    protected $touches = ['post'];

    /**
     * Get the post that the comment belongs to.
     */
    public function post()
    {
        return $this->belongsTo(Post::class);
    }
}
```