Course - Laravel Framework

Eloquent ORM

Laravel bao gồm Eloquent, một chương trình bố trí quan hệ đối tượng (ORM) làm cho việc tương tác với cơ sở dữ liệu của bạn trở nên thú vị. Khi sử dụng Eloquent, mỗi bảng cơ sở dữ liệu có một "Model" tương ứng được sử dụng để tương tác với bảng đó. Ngoài việc truy xuất các record từ bảng cơ sở dữ liệu, các mô hình Eloquent cho phép bạn chèn, cập nhật và xóa các record khỏi bảng.

Tags: eloquent orm, quan he du lieu, laravel

Giới thiệu

Laravel bao gồm Eloquent, một chương trình bố trí quan hệ đối tượng (ORM) làm cho việc tương tác với cơ sở dữ liệu của bạn trở nên thú vị. Khi sử dụng Eloquent, mỗi bảng cơ sở dữ liệu có một "Model" tương ứng được sử dụng để tương tác với bảng đó. Ngoài việc truy xuất các record từ bảng cơ sở dữ liệu, các mô hình Eloquent cho phép bạn chèn, cập nhật và xóa các record khỏi bảng.

> Trước khi bắt đầu, hãy đảm bảo cấu hình kết nối cơ sở dữ liệu trong tập tin cấu hình *config/database.php* của ứng dụng của bạn. Để biết thêm thông tin về cách cấu hình cơ sở dữ liệu của bạn, hãy xem tài liệu cấu hình cơ sở dữ liệu.

Tạo ra các class mô hình

Để bắt đầu, hãy tạo một model Eloquent. Các model thường nằm trong thư mục *app\Models* và mở rộng class `Illuminate\Database\Eloquent\Model`. Bạn có thể sử dụng lệnh Artisan `make:model` để tạo một model mới:

```
php artisan make:model Flight
```

Nếu bạn muốn tạo migration cơ sở dữ liệu khi tạo model, bạn có thể sử dụng tùy chọn `--migration` hoặc `-m`:

```
php artisan make:model Flight --migration
```

Bạn có thể tạo ra nhiều loại class khác nhau khi tạo một model, chẳng hạn như factory, seeder, policy, controller và form request. Ngoài ra, các tùy chọn này có thể được kết hợp để tạo nhiều class cùng một lúc:

```
# Generate a model and a FlightFactory class...
php artisan make:model Flight --factory
php artisan make:model Flight -f

# Generate a model and a FlightSeeder class...
php artisan make:model Flight --seed
php artisan make:model Flight -s

# Generate a model and a FlightController class...
```

```
php artisan make:model Flight --controller
php artisan make:model Flight -c

# Generate a model, FlightController resource class, and form request classes...
php artisan make:model Flight --controller --resource --requests
php artisan make:model Flight -crR

# Generate a model and a FlightPolicy class...
php artisan make:model Flight --policy

# Generate a model and a migration, factory, seeder, and controller...
php artisan make:model Flight -mfsc

# Shortcut to generate a model, migration, factory, seeder, policy, controller, and form requests
php artisan make:model Flight --all

# Generate a pivot model...
php artisan make:model Member --pivot
```

Các quy ước model

Các model được tạo bởi lệnh `make:model` sẽ được đặt trong thư mục *app/Models*. Hãy xem xét một class model cơ bản và thảo luận một số quy ước chính của Eloquent:

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

class Flight extends Model
{
  //
}
```

Tên bảng

Sau khi xem qua ví dụ trên, bạn có thể nhận thấy rằng chúng tôi đã không cho Eloquent biết

bảng cơ sở dữ liệu nào tương ứng với model `Flight` của chúng tôi. Theo quy ước, "trường hợp con rắn", tên số nhiều của class sẽ được sử dụng làm tên bảng trừ khi một tên khác được chỉ định rõ ràng. Vì vậy, trong trường hợp này, Eloquent sẽ giả sử model `Flight` lưu trữ record trong bảng `flights`, trong khi model `AirTrafficController` sẽ lưu trữ record trong bảng `air_traffic_controllers`.

Nếu bảng cơ sở dữ liệu tương ứng của model của bạn không phù hợp với quy ước này, bạn có thể chỉ định tên bảng của model theo cách thủ công bằng cách xác định thuộc tính `table` trên model:

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

class Flight extends Model
{
  /**
   * The table associated with the model.
   *
   * @var string
   */
  protected $table = 'my_flights';
}
```

Khóa chính

Eloquent cũng sẽ giả định rằng bảng cơ sở dữ liệu tương ứng của mỗi model có một cột khóa chính có tên là `id`. Nếu cần, bạn có thể xác định thuộc tính *protected* `$primaryKey` trên model của mình để chỉ định một cột khác đóng vai trò là khóa chính của model của bạn:

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

class Flight extends Model
{
```

```
    /**
     * The primary key associated with the table.
     *
     * @var string
     */
    protected $primaryKey = 'flight_id';
}
```

Ngoài ra, Eloquent giả định rằng khóa chính là một giá trị số nguyên tăng dần, có nghĩa là Eloquent sẽ tự động chuyển khóa chính thành một số nguyên. Nếu bạn muốn sử dụng khóa chính không tăng hoặc không phải số, bạn phải xác định thuộc tính public `$incrementing` trên model của bạn được đặt thành `false`:

```
<?php

class Flight extends Model
{
    /**
     * Indicates if the model's ID is auto-incrementing.
     *
     * @var bool
     */
    public $incrementing = false;
}
```

Nếu khóa chính của model của bạn không phải là số nguyên, bạn nên xác định thuộc tính *protected* `$keyType` trên model của mình. Thuộc tính này phải có giá trị là `string`:

```
<?php

class Flight extends Model
{
    /**
     * The data type of the auto-incrementing ID.
     *
     * @var string
     */
```

```
    protected $keyType = 'string';
}
```

Khóa chính "composite"

Eloquent yêu cầu mỗi model phải có ít nhất một "ID" nhận dạng duy nhất có thể dùng làm khóa chính của nó. Các khóa chính "hỗn hợp" không được hỗ trợ bởi các model Eloquent. Tuy nhiên, bạn có thể tự do thêm các chỉ mục nhiều cột, duy nhất vào bảng cơ sở dữ liệu của mình ngoài khóa chính nhận dạng duy nhất của bảng.

Timestamps

Theo mặc định, Eloquent mong đợi các cột `created_at` và `updated_at` tồn tại trên bảng cơ sở dữ liệu tương ứng của model của bạn. Eloquent sẽ tự động đặt các giá trị của cột này khi các model được tạo hoặc cập nhật. Nếu bạn không muốn các cột này được quản lý tự động bởi Eloquent, bạn nên xác định thuộc tính `$timestamps` trên model của mình với giá trị là `false`:

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

class Flight extends Model
{
  /**
   * Indicates if the model should be timestamped.
   *
   * @var bool
   */
  public $timestamps = false;
}
```

Nếu bạn cần tùy chỉnh định dạng thời gian của model, hãy thiết lập thuộc tính `$dateFormat` trên model của bạn. Thuộc tính này xác định cách các thuộc tính ngày tháng được lưu trữ trong cơ sở dữ liệu cũng như định dạng của chúng khi mô hình được tuần tự hóa thành một mảng hoặc JSON:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
  /**
   * The storage format of the model's date columns.
   *
   * @var string
   */
  protected $dateFormat = 'U';
}
```

Nếu bạn cần tùy chỉnh tên của các cột được sử dụng để lưu trữ thời gian, bạn có thể xác định các hằng số `CREATED_AT` và `UPDATED_AT` trên model của mình:

```php
<?php

class Flight extends Model
{
  const CREATED_AT = 'creation_date';
  const UPDATED_AT = 'updated_date';
}
```

Các kết nối database

Theo mặc định, tất cả các mô hình Eloquent sẽ sử dụng kết nối cơ sở dữ liệu mặc định được cấu hình cho ứng dụng của bạn. Nếu bạn muốn chỉ định một kết nối khác sẽ được sử dụng khi tương tác với một model cụ thể, bạn nên xác định thuộc tính `$connection` trên model:

```php
<?php
namespace App\Models;
```

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

class Flight extends Model
{
  /**
   * The database connection that should be used by the model.
   *
   * @var string
   */
  protected $connection = 'sqlite';
}
```

Các giá trị thuộc tính mặc định

Theo mặc định, một đối tượng model mới được khởi tạo sẽ không chứa bất kỳ giá trị thuộc tính attribute nào. Nếu bạn muốn xác định các giá trị mặc định cho một số thuộc tính attribute của model, bạn có thể xác định thuộc tính `$attributes` trên đối tượng model của mình:

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

class Flight extends Model
{
  /**
   * The model's default values for attributes.
   *
   * @var array
   */
  protected $attributes = [
    'delayed' => false,
  ];
}
```

Nhận các model

Khi bạn đã tạo một model và bảng cơ sở dữ liệu liên quan của nó, bạn đã sẵn sàng bắt đầu truy xuất dữ liệu từ cơ sở dữ liệu của mình. Bạn có thể coi mỗi mô hình Eloquent như một query builder mạnh mẽ cho phép bạn truy vấn thoải mái bảng cơ sở dữ liệu được liên kết với model. Phương thức `all` của model sẽ truy xuất tất cả các record từ bảng cơ sở dữ liệu được liên kết với model:

```
use App\Models\Flight;

foreach (Flight::all() as $flight) {
  echo $flight->name;
}
```

Tạo các truy vấn

Phương thức Eloquent `all` sẽ trả về tất cả các kết quả trong bảng của model. Tuy nhiên, vì mỗi model Eloquent đóng vai trò như một query builder, bạn có thể thêm các ràng buộc bổ sung vào các truy vấn và sau đó gọi phương thức `get` để truy xuất kết quả:

```
$flights = Flight::where('active', 1)
  ->orderBy('name')
  ->take(10)
  ->get();
```

> Vì các model Eloquent là query builder, bạn nên xem lại tất cả các phương thức được cung cấp query builder của Laravel. Bạn có thể sử dụng bất kỳ phương thức nào trong số những phương thức này khi viết các truy vấn Eloquent của mình.

Làm mới các model

Nếu bạn đã có một đối tượng của model Eloquent được truy xuất từ cơ sở dữ liệu, bạn có thể "làm mới" model bằng các phương thức `fresh` và `refresh`. Phương thức `fresh` sẽ lấy lại model từ cơ sở dữ liệu. Đối tượng model hiện tại sẽ không bị ảnh hưởng:

```php
$flight = Flight::where('number', 'FR 900')->first();

$freshFlight = $flight->fresh();
```

Phương thức `refresh` sẽ làm mới lại model hiện có bằng cách dữ liệu mới từ cơ sở dữ liệu. Ngoài ra, tất cả các mối quan hệ đã tải được của nó cũng sẽ được làm mới:

```php
$flight = Flight::where('number', 'FR 900')->first();
$flight->number = 'FR 456';
$flight->refresh();
$flight->number; // "FR 900"
```

Các collection

Như chúng ta đã thấy, các phương thức Eloquent như all và get sẽ truy xuất nhiều record từ cơ sở dữ liệu. Tuy nhiên, các phương thức này không trả về một mảng PHP thuần túy. Thay vào đó, một đối tượng của `Illuminate\Database\Eloquent\Collection` được trả về.

Class Eloquent `Collection` mở rộng class `Illuminate\Support\Collection` cơ sở của Laravel, class này cung cấp nhiều phương thức hữu ích để tương tác với các tập hợp dữ liệu. Ví dụ, phương thức `reject` có thể được sử dụng để loại bỏ các model khỏi bộ sưu tập dựa trên kết quả của một lần đóng được gọi:

```php
$flights = Flight::where('destination', 'Paris')->get();

$flights = $flights->reject(function ($flight) {
  return $flight->cancelled;
});
```

Ngoài các phương thức được cung cấp bởi class collection cơ sở của Laravel, class collection Eloquent cung cấp một số phương thức bổ sung dành riêng cho việc tương tác với collection của các mô hình Eloquent.

Vì tất cả các collection của Laravel đều triển khai các giao diện có thể lặp lại của PHP, bạn có thể lặp lại các collection như thể chúng là một mảng:

```php
foreach ($flights as $flight) {
  echo $flight->name;
}
```

Phân nhỏ kết quả

Ứng dụng của bạn có thể hết bộ nhớ nếu bạn cố gắng tải hàng chục nghìn record Eloquent thông qua các phương thức `all` hoặc `get`. Thay vì sử dụng các phương thức này, phương thức `chunk` có thể được sử dụng để xử lý số lượng lớn các model hiệu quả hơn.

Phương thức `chunk` sẽ lấy một tập hợp con của các model Eloquent, truyền chúng đến một hàm xử lý để xử lý. Vì chỉ một phần nhỏ hiện có nào đó của các model Eloquent được truy xuất tại một thời điểm, phương thức `chunk` sẽ giảm đáng kể mức sử dụng bộ nhớ khi làm việc với một số lượng lớn các model:

```php
use App\Models\Flight;

Flight::chunk(200, function ($flights) {
  foreach ($flights as $flight) {
    //
  }
});
```

Đối số đầu tiên được truyền cho phương thức `chunk` là số lượng record bạn muốn nhận trên mỗi "chunk". Hàm xử lý được truyền làm đối số thứ hai sẽ được gọi cho mỗi đoạn được truy xuất từ cơ sở dữ liệu. Một truy vấn cơ sở dữ liệu sẽ được thực hiện để truy xuất từng đoạn record được chuyển đến hàm xử lý.

Nếu bạn đang lọc kết quả của phương thức `chunk` dựa trên một cột mà bạn cũng sẽ cập nhật trong khi lặp lại các kết quả, bạn nên sử dụng phương thức `chunkById`. Sử dụng phương thức `chunk` trong các trường hợp này có thể dẫn đến kết quả không mong muốn và không nhất quán. Bên trong, phương thức `chunkById` sẽ luôn truy xuất các model có cột `id` lớn hơn model cuối cùng trong chunk trước đó:

```php
Flight::where('departed', true)
  ->chunkById(200, function ($flights) {
    $flights->each->update(['departed' => false]);
```

```
    }, $column = 'id');
```

Tải dữ liệu tự động theo từng đoạn

Phương thức `lazy` hoạt động tương tự như phương thức `chunk` theo nghĩa là, đằng sau, nó thực hiện truy vấn theo từng phần. Tuy nhiên, thay vì truyền trực tiếp từng đoạn vào một lệnh callback, phương thức `lazy` trả về một mô hình `LazyCollection` của Eloquent được làm phẳng, cho phép bạn tương tác với các kết quả dưới dạng một luồng duy nhất:

```
use App\Models\Flight;

foreach (Flight::lazy() as $flight) {
    //
}
```

Nếu bạn đang lọc kết quả của phương thức `lazy` dựa trên một cột mà bạn cũng sẽ cập nhật trong khi lặp qua các kết quả, bạn nên sử dụng phương thức `lazyById`. Bên trong, phương thức `lazyById` sẽ luôn truy xuất các model có cột `id` lớn hơn model cuối cùng trong đoạn trước:

```
Flight::where('departed', true)
    ->lazyById(200, $column = 'id')
    ->each->update(['departed' => false]);
```

Bạn có thể lọc kết quả dựa trên thứ tự giảm dần của `id` bằng phương thức `lazyByIdDesc`.

Truy vấn Cursors

Tương tự như phương thức `lazy`, phương thức `cursor` có thể được sử dụng để giảm đáng kể mức tiêu thụ bộ nhớ của ứng dụng khi lặp qua hàng chục nghìn record của Eloquent model.

Phương thức `cursor` sẽ chỉ thực hiện một truy vấn cơ sở dữ liệu duy nhất; tuy nhiên, các Eloquent model riêng lẻ sẽ không bị chồng chéo cho đến khi chúng thực sự được lặp qua. Do đó, chỉ có một model Eloquent được lưu trong bộ nhớ tại bất kỳ thời điểm nào trong khi lặp.

Vì phương thức `cursor` chỉ lưu giữ một model Eloquent duy nhất trong bộ nhớ tại một thời điểm, nên nó không thể thiết lập các mối quan hệ tải. Thay vào đó, nếu bạn cần tải các mối quan hệ một cách háo hức, hãy cân nhắc sử dụng phương thức `lazy`.

Bên trong, phương thức `cursor` sử dụng PHP generators để triển khai chức năng này:

```
use App\Models\Flight;

foreach (Flight::where('destination', 'Zurich')->cursor() as $flight) {
  //
}
```

Phương thức `cursor` trả về một đối tượng `Illuminate\Support\LazyCollection`. Lazy collection cho phép bạn sử dụng nhiều phương thức collection có sẵn trên các Laravel collection điển hình trong khi chỉ tải một model duy nhất vào bộ nhớ tại một thời điểm:

```
use App\Models\User;

$users = User::cursor()->filter(function ($user) {
  return $user->id > 500;
});

foreach ($users as $user) {
  echo $user->id;
}
```

Mặc dù phương thức cursor sử dụng ít bộ nhớ hơn nhiều so với truy vấn thông thường (bằng cách chỉ giữ một Eloquent model duy nhất trong bộ nhớ tại một thời điểm), nhưng cuối cùng nó vẫn sẽ hết bộ nhớ. Điều này là do driver PDO của PHP trong bộ đệm ẩn tất cả các kết quả truy vấn thô trong bộ đệm của nó. Nếu bạn đang xử lý một số lượng rất lớn các record Eloquent, hãy xem xét sử dụng phương thức `lazy` thay thế.

Các truy vấn con

Select trong truy vấn con

Eloquent cũng cung cấp hỗ trợ truy vấn con, cho phép bạn lấy thông tin từ các bảng liên

quan trong một truy vấn duy nhất. Ví dụ, hãy tưởng tượng rằng chúng ta có một bảng `destinations` của chuyến bay và một bảng `flights` đến các điểm đến. Bảng `flights` chứa một cột `arrived_at` cho biết thời điểm chuyến bay đến điểm đến.

Sử dụng chức năng truy vấn phụ có sẵn cho phương thức `select` và `addSelect` của query builder, chúng tôi có thể chọn tất cả các `destinations` và tên của chuyến bay đã đến điểm đến đó gần đây nhất bằng cách sử dụng một truy vấn:

```
use App\Models\Destination;
use App\Models\Flight;

return Destination::addSelect(['last_flight' => Flight::select('name')
  ->whereColumn('destination_id', 'destinations.id')
  ->orderByDesc('arrived_at')
  ->limit(1)
])->get();
```

Sắp xếp thứ tự trong truy vấn con

Ngoài ra, hàm `orderBy` của query builder hỗ trợ các truy vấn con. Tiếp tục sử dụng ví dụ về chuyến bay của chúng ta, chúng ta có thể sử dụng chức năng này để sắp xếp tất cả các điểm đến dựa trên thời điểm chuyến bay cuối cùng đến điểm đến đó. Một lần nữa, điều này có thể được thực hiện trong khi thực hiện một truy vấn cơ sở dữ liệu duy nhất:

```
return Destination::orderByDesc(
  Flight::select('arrived_at')
    ->whereColumn('destination_id', 'destinations.id')
    ->orderByDesc('arrived_at')
    ->limit(1)
)->get();
```

Truy xuất model đơn lẻ hoặt số thống kê

Ngoài việc truy xuất tất cả các record phù hợp với một truy vấn nhất định, bạn cũng có thể truy xuất các record đơn lẻ bằng cách sử dụng các phương thức `find`, `first` hoặc `firstWhere`. Thay vì trả về một collection các model, các phương thức này trả về một đối tượng model duy nhất:

```php
use App\Models\Flight;

// Retrieve a model by its primary key...
$flight = Flight::find(1);

// Retrieve the first model matching the query constraints...
$flight = Flight::where('active', 1)->first();

// Alternative to retrieving the first model matching the query constraints...
$flight = Flight::firstWhere('active', 1);
```

Đôi khi bạn có thể muốn truy xuất kết quả đầu tiên của một truy vấn hoặc thực hiện một số hành động khác nếu không có kết quả nào được tìm thấy. Phương thức `firstOr` sẽ trả về kết quả đầu tiên phù hợp với truy vấn hoặc nếu không tìm thấy kết quả nào, hãy thực hiện hàm xử lý đã cho. Giá trị được trả về bởi hàm xử lý sẽ được coi là kết quả của phương thức `firstOr`:

```php
$model = Flight::where('legs', '>', 3)->firstOr(function () {
  // ...
});
```

Ngoại lệ Not Found

Đôi khi bạn có thể muốn đưa ra một ngoại lệ nếu không tìm thấy một model. Điều này đặc biệt hữu ích trong các route hoặc controller. Phương thức `findOrFail` và `firstOrFail` sẽ truy xuất kết quả đầu tiên của truy vấn; tuy nhiên, nếu không tìm thấy kết quả nào, một `Illuminate\Database\Eloquent\ModelNotFoundException` sẽ được đưa ra:

```php
$flight = Flight::findOrFail(1);

$flight = Flight::where('legs', '>', 3)->firstOrFail();
```

Nếu `ModelNotFoundException` không được bắt, phản hồi HTTP 404 sẽ tự động được gửi lại cho máy khách:

```php
use App\Models\Flight;

Route::get('/api/flights/{id}', function ($id) {
  return Flight::findOrFail($id);
});
```

Truy xuất hoặt tạo model

Phương thức `firstOrCreate` sẽ cố gắng xác định một record cơ sở dữ liệu bằng cách sử dụng các cặp cột/giá trị đã cho. Nếu không thể tìm thấy model trong cơ sở dữ liệu, một record sẽ được chèn với các đối số đã cho:

Phương thức `firstOrNew`, giống như `firstOrCreate`, sẽ cố gắng định vị một record trong cơ sở dữ liệu khớp với các đối số đã cho. Tuy nhiên, nếu một model không được tìm thấy, một đối tượng model mới sẽ được trả về. Lưu ý rằng model do `firstOrNew` trả về vẫn chưa được lưu vào cơ sở dữ liệu. Bạn sẽ cần phải gọi phương thức `save` theo cách thủ công để lưu trữ nó:

```php
use App\Models\Flight;

// Retrieve flight by name or create it if it doesn't exist...
$flight = Flight::firstOrCreate([
    'name' => 'London to Paris'
]);

// Retrieve flight by name or create it with the name, delayed, and arrival_time attributes...
$flight = Flight::firstOrCreate(
    ['name' => 'London to Paris'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);

// Retrieve flight by name or instantiate a new Flight instance...
$flight = Flight::firstOrNew([
    'name' => 'London to Paris'
]);

// Retrieve flight by name or instantiate with the name, delayed, and arrival_time attributes...
```

```php
$flight = Flight::firstOrNew(
    ['name' => 'Tokyo to Sydney'],
    ['delayed' => 1, 'arrival_time' => '11:30']
);
```

Truy xuất số liệu thông kê

Khi tương tác với các model Eloquent, bạn cũng có thể sử dụng phương thức `count`, `sum`, `max` và các phương thức tổng hợp khác được cung cấp bởi query builder Laravel. Như bạn có thể mong đợi, các phương thức này trả về một giá trị vô hướng thay vì một đối tượng model Eloquent:

```php
$count = Flight::where('active', 1)->count();

$max = Flight::where('active', 1)->max('price');
```

Các mô hình chèn và cập nhật

Chèn dữ liệu

Tất nhiên, khi sử dụng Eloquent, chúng ta không chỉ cần lấy các model từ cơ sở dữ liệu. Chúng tôi cũng cần phải chèn các record mới. Rất may, Eloquent làm cho nó trở nên đơn giản. Để chèn một record mới vào cơ sở dữ liệu, bạn nên khởi tạo một đối tượng model mới và thiết lập các thuộc tính trên model. Sau đó, gọi phương thức `save` trên đối tượng model:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Models\Flight;
use Illuminate\Http\Request;

class FlightController extends Controller
{
```

```
    /**
     * Store a new flight in the database.
     *
     * @param  \Illuminate\Http\Request  $request
     * @return \Illuminate\Http\Response
     */
    public function store(Request $request)
    {
        // Validate the request...
        $flight = new Flight;
        $flight->name = $request->name;
        $flight->save();
    }
}
```

Trong ví dụ này, chúng tôi gán trường `name` từ yêu cầu HTTP request đến cho thuộc tính `name` của đối tượng model `App\Models\Flight`. Khi chúng ta gọi phương thức `save`, một record sẽ được chèn vào cơ sở dữ liệu. Dấu thời gian `created_at` và `updated_at` của model sẽ tự động được đặt khi phương thức `save` được gọi, vì vậy không cần đặt chúng theo cách thủ công.

Ngoài ra, bạn có thể sử dụng phương thức `create` để "lưu" một model mới bằng cách sử dụng một câu lệnh PHP. Đối tượng model được chèn sẽ được trả lại cho bạn bằng phương thức `create`:

```
use App\Models\Flight;

$flight = Flight::create([
    'name' => 'London to Paris',
]);
```

Tuy nhiên, trước khi sử dụng phương thức `create`, bạn sẽ cần chỉ định thuộc tính có thể điền hoặc được bảo vệ trên class model của mình. Các thuộc tính này là bắt buộc vì tất cả các Eloquent model được bảo vệ chống lại các lỗ hổng phân công hàng loạt theo mặc định. Để tìm hiểu thêm về bài tập khối lượng, mời các bạn tham khảo tài liệu bài tập khối lượng lớn.

Cập nhật

Phương thức `save` cũng có thể được sử dụng để cập nhật các model đã tồn tại trong cơ sở dữ liệu. Để cập nhật một model, bạn nên truy xuất model đó và đặt bất kỳ thuộc tính nào bạn muốn cập nhật. Sau đó, bạn nên gọi phương thức `save` của model. Một lần nữa, dấu thời gian `updated_at` sẽ tự động được cập nhật, vì vậy không cần phải đặt giá trị của nó theo cách thủ công:

```
use App\Models\Flight;

$flight = Flight::find(1);
$flight->name = 'Paris to London';
$flight->save();
```

Cập nhật hàng loạt

Cập nhật cũng có thể được thực hiện đối với các model có cùng một truy vấn nào đó. Trong ví dụ này, tất cả các `flights` đang hoạt động và có `destination` là `San Diego` sẽ được đánh dấu là bị hoãn:

```
Flight::where('active', 1)
  ->where('destination', 'San Diego')
  ->update(['delayed' => 1]);
```

Phương thức `update` yêu cầu một mảng các cặp cột và giá trị đại diện cho các cột cần được cập nhật. Phương thức `update` trả về số lượng hàng bị ảnh hưởng.

> **Chú ý:** Khi cập nhật hàng loạt qua Eloquent, các event của model như saving, saved, updating và updated sẽ không được kích hoạt cho các model đã cập nhật. Điều này là do các model không bao giờ thực sự được truy xuất khi cập nhật hàng loạt.

Kiểm tra các thay đổi thuộc tính

Eloquent cung cấp các phương thức `isDirty`, `isClean` và `wasChanged` để kiểm tra trạng thái bên trong của model của bạn và xác định các thuộc tính của nó đã thay đổi như thế nào so với khi model được truy xuất ban đầu.

Phương thức `isDirty` xác định xem có bất kỳ thuộc tính nào của model bị thay đổi kể từ

khi model được truy xuất hay không. Bạn có thể truyền một tên thuộc tính cụ thể cho phương thức `isDirty` để xác định xem một thuộc tính cụ thể có bị cài hay không. `IsClean` sẽ xác định xem một thuộc tính không thay đổi kể từ khi mô hình được truy xuất hay không. Phương thức này cũng chấp nhận một đối số thuộc tính tùy chọn:

```
use App\Models\User;

$user = User::create([
  'first_name' => 'Taylor',
  'last_name' => 'Otwell',
  'title' => 'Developer',
]);

$user->title = 'Painter';

$user->isDirty(); // true
$user->isDirty('title'); // true
$user->isDirty('first_name'); // false

$user->isClean(); // false
$user->isClean('title'); // false
$user->isClean('first_name'); // true

$user->save();

$user->isDirty(); // false
$user->isClean(); // true
```

Phương thức `wasChanged` xác định xem có bất kỳ thuộc tính nào bị thay đổi khi model được lưu lần cuối cùng trong chu kỳ yêu cầu hiện tại hay không. Nếu cần, bạn có thể truyền một tên thuộc tính để xem liệu một thuộc tính cụ thể có bị thay đổi hay không:

```
$user = User::create([
  'first_name' => 'Taylor',
  'last_name' => 'Otwell',
  'title' => 'Developer',
]);
```

```php
$user->title = 'Painter';

$user->save();

$user->wasChanged(); // true
$user->wasChanged('title'); // true
$user->wasChanged('first_name'); // false
```

Phương thức `getOriginal` trả về một mảng chứa các thuộc tính ban đầu của model bất kể bất kỳ thay đổi nào đối với model kể từ khi nó được truy xuất. Nếu cần, bạn có thể truyền một tên thuộc tính cụ thể để nhận giá trị ban đầu của một thuộc tính cụ thể:

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

$user->name; // John
$user->email; // john@example.com

$user->name = "Jack";
$user->name; // Jack

$user->getOriginal('name'); // John
$user->getOriginal(); // Array of original attributes...
```

Phân bổ hàng loạt

Bạn có thể sử dụng phương thức `create` để "lưu" một model mới bằng cách sử dụng một câu lệnh PHP. Đối tượng model được chèn sẽ được trả lại cho bạn theo phương thức:

```php
use App\Models\Flight;

$flight = Flight::create([
  'name' => 'London to Paris',
]);
```

Tuy nhiên, trước khi sử dụng phương thức `create`, bạn sẽ cần chỉ định thuộc tính fillable

hoặc guarded trên class model của mình. Các thuộc tính này là bắt buộc vì tất cả các Eloquent model được bảo vệ chống lại các lỗ hổng phân bổ hàng loạt theo mặc định.

Lỗ hổng phân bổ hàng loạt xảy ra khi người dùng truyền một trường yêu cầu HTTP không mong muốn và trường đó thay đổi một cột trong cơ sở dữ liệu của bạn mà bạn không mong đợi. Ví dụ: một người dùng độc hại có thể gửi một tham số `is_admin` thông qua một yêu cầu HTTP request, sau đó được truyền vào phương thức create của model của bạn, cho phép người dùng tự chuyển lên quản trị viên.

Vì vậy, để bắt đầu, bạn nên xác định thuộc tính model nào bạn muốn để có thể gán hàng loạt. Bạn có thể làm điều này bằng cách sử dụng thuộc tính `$fillable` trên model. Ví dụ: hãy đặt thuộc tính `name` của model `Flight` có thể gán hàng loạt:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
  /**
   * The attributes that are mass assignable.
   *
   * @var array
   */
  protected $fillable = ['name'];
}
```

Khi bạn đã chỉ định thuộc tính nào có thể gán hàng loạt, bạn có thể sử dụng phương thức `create` để chèn một bản ghi mới vào cơ sở dữ liệu. Phương thức `create` trả về cá thể model mới được tạo:

```php
$flight = Flight::create(['name' => 'London to Paris']);
```

Nếu bạn đã có một đối tượng model, bạn có thể sử dụng phương thức `fill` vào để điền vào nó một mảng thuộc tính attribute:

```php
$flight->fill(['name' => 'Amsterdam to Frankfurt']);
```

Gán hàng loạt vào các cột JSON

Khi gán các cột JSON, khóa có thể gán hàng loạt của mỗi cột phải được chỉ định trong mảng `$fillable` của model của bạn. Để bảo mật, Laravel không hỗ trợ cập nhật các thuộc tính JSON lồng nhau khi sử dụng thuộc tính `guarded`:

```php
/**
 * The attributes that are mass assignable.
 *
 * @var array
 */
protected $fillable = [
  'options->enabled',
];
```

Cho phép gán hàng loạt

Nếu bạn muốn gán hàng loạt các thuộc tính của mình, bạn có thể xác định thuộc tính `$guarded` của model của bạn là một mảng trống. Nếu bạn chọn không theo dõi model của mình, bạn nên đặc biệt lưu ý để luôn tạo thủ công các mảng được truyền cho các phương thức `fill`, `create` và `update` của Eloquent:

```php
/**
 * The attributes that aren't mass assignable.
 *
 * @var array
 */
protected $guarded = [];
```

Vừa tạo vừa cập nhật

Đôi khi, bạn có thể cần cập nhật một model hiện có hoặc tạo một model mới nếu không có model phù hợp nào tồn tại. Giống như phương thức `firstOrCreate`, phương thức

`updateOrCreate` trong model, vì vậy không cần phải gọi phương thức `save` theo cách thủ công.

Trong ví dụ dưới đây, nếu một chuyến bay tồn tại với vị trí khởi hành `departure` là `Oakland` và vị trí `destination` là `San Diego`, các cột giá `price` và chiết khấu `discounted` của chuyến bay đó sẽ được cập nhật. Nếu không có chuyến bay nào như vậy tồn tại, một chuyến bay mới sẽ được tạo với các thuộc tính là kết quả của việc tổng hợp mảng đối số đầu tiên với mảng đối số thứ hai:

```
$flight = Flight::updateOrCreate(
  ['departure' => 'Oakland', 'destination' => 'San Diego'],
  ['price' => 99, 'discounted' => 1]
);
```

Nếu bạn muốn thực hiện nhiều thao tác "upsert" trong một truy vấn, thì bạn nên sử dụng phương thức `upsert` thay thế. Đối số đầu tiên của phương thức bao gồm các giá trị để chèn hoặc cập nhật, trong khi đối số thứ hai liệt kê (các) cột xác định duy nhất các bản ghi trong bảng được liên kết. Đối số thứ ba và cuối cùng của phương thức là một mảng các cột cần được cập nhật nếu một record phù hợp đã tồn tại trong cơ sở dữ liệu. Phương thức `upert` sẽ tự động đặt dấu thời gian `created_at` và `updated_at` nếu dấu thời gian được bật trên model:

```
Flight::upsert([
  ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
  ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
], ['departure', 'destination'], ['price']);
```

Xóa model

Để xóa một model, bạn có thể gọi phương thức `delete` trên đối tượng model:

```
use App\Models\Flight;

$flight = Flight::find(1);

$flight->delete();
```

Bạn có thể gọi phương thức `truncate` để xóa tất cả các record cơ sở dữ liệu liên quan của model. Thao tác *truncate* cũng sẽ làm mới mọi ID tự động tăng dần trở về zero trên bảng được liên kết của model:

```
Flight::truncate();
```

Xóa một model đã có bằng khóa chính của nó

Trong ví dụ trên, chúng tôi đang truy xuất mô hình từ cơ sở dữ liệu trước khi gọi phương thức `delete`. Tuy nhiên, nếu bạn biết khóa chính của model, bạn có thể xóa model mà không cần truy xuất rõ ràng bằng cách gọi phương thức `destroy`. Ngoài việc chấp nhận một khóa chính, phương thức `destroy` sẽ chấp nhận nhiều khóa chính, một mảng khóa chính hoặc một tập hợp các khóa chính:

```
Flight::destroy(1);

Flight::destroy(1, 2, 3);

Flight::destroy([1, 2, 3]);

Flight::destroy(collect([1, 2, 3]));
```

> **Chú ý:** Phương thức `destroy` tải từng mô hình riêng lẻ và gọi phương thức `delete` để các event `deleting` và `deleted` được gửi đúng cách cho từng model.

Xóa các model bằng cách sử dụng các truy vấn

Tất nhiên, bạn có thể tạo một truy vấn Eloquent để xóa tất cả các model phù hợp với tiêu chí truy vấn của bạn. Trong ví dụ này, chúng tôi sẽ xóa tất cả các chuyến bay được đánh dấu là không hoạt động. Giống như cập nhật hàng loạt, xóa hàng loạt sẽ không gửi các model event cho các model đã bị xóa:

```
$deleted = Flight::where('active', 0)->delete();
```

> **Chú ý:** Khi thực hiện câu lệnh xóa hàng loạt qua Eloquent, các event của model `deleting` và `deleted` sẽ không được gửi cho các model đã xóa. Điều này là do các

model không bao giờ thực sự được truy xuất khi thực hiện câu lệnh xóa.

Xóa mềm

Ngoài việc thực sự xóa các bản ghi khỏi cơ sở dữ liệu của bạn, Eloquent cũng có thể thực hiện các model "xóa mềm". Khi các mô hình bị xóa mềm, chúng không thực sự bị xóa khỏi cơ sở dữ liệu của bạn. Thay vào đó, thuộc tính `deleted_at` được đặt trên model cho biết ngày và giờ tại thời điểm đó model bị "xóa". Để bật tính năng xóa mềm cho một model, hãy thêm trait `Illuminate\Database\Eloquent\SoftDeletes` vào model:

```php
<?php

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

class Flight extends Model
{
  use SoftDeletes;
}
```

Trait `SoftDeletes` sẽ tự động chuyển thuộc tính `deleted_at` thành đối tượng `DateTime/Carbon` cho bạn.

Bạn cũng nên thêm cột `deleted_at` vào bảng cơ sở dữ liệu của mình. Schema builder của Laravel chứa một phương thức giúp để tạo cột này:

```php
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

Schema::table('flights', function (Blueprint $table) {
  $table->softDeletes();
});

Schema::table('flights', function (Blueprint $table) {
  $table->dropSoftDeletes();
});
```

Bây giờ, khi bạn gọi phương thức `delete` trên model, cột `deleted_at` sẽ được đặt thành ngày và giờ hiện tại. Tuy nhiên, record cơ sở dữ liệu của model sẽ được để lại trong bảng. Khi truy vấn một model sử dụng tính năng xóa mềm, các model đã xóa mềm sẽ tự động bị loại trừ khỏi tất cả các kết quả truy vấn.

Để xác định xem một đối tượng model nhất định đã bị xóa mềm hay chưa, bạn có thể sử dụng phương thức `trashed`:

```php
if ($flight->trashed()) {
  //
}
```

Khôi phục các model đã bị xóa mềm

Đôi khi bạn có thể muốn "un-delete" một model đã xóa mềm. Để khôi phục một model đã xóa mềm, bạn có thể gọi phương thức `restore` trên một đối tượng model. Phương thức `restore` sẽ đặt cột `deleted_at` của model thành `null`:

```php
$flight->restore();
```

Bạn cũng có thể sử dụng phương thức `restore` trong một truy vấn để khôi phục nhiều model. Giống như các thao tác thay đổi "hàng loạt" khác, thao tác này sẽ không gửi bất kỳ event của model nào cho các model được khôi phục:

```php
Flight::withTrashed()
```

```
    ->where('airline_id', 1)
    ->restore();
```

Phương thức `restore` cũng có thể được sử dụng khi xây dựng các truy vấn mối quan hệ:

```
$flight->history()->restore();
```

Xóa vĩnh viễn các mô hình

Đôi khi bạn có thể cần thực sự xóa một model khỏi cơ sở dữ liệu của mình. Bạn có thể sử dụng phương thức `forceDelete` để xóa vĩnh viễn một model đã xóa mềm khỏi bảng cơ sở dữ liệu:

```
$flight->forceDelete();
```

Bạn cũng có thể sử dụng phương thức `forceDelete` khi xây dựng các truy vấn mối quan hệ Eloquent:

```
$flight->history()->forceDelete();
```

Truy vấn các model bị xóa mềm

Chọn các model bị xóa mềm

Như đã lưu ý ở trên, các model bị xóa mềm sẽ tự động bị loại trừ khỏi kết quả truy vấn. Tuy nhiên, bạn cũng có thể đưa các mô hình đã xóa mềm vào kết quả của truy vấn bằng cách gọi phương thức `withTrashed` trên truy vấn:

```
use App\Models\Flight;

$flights = Flight::withTrashed()
    ->where('account_id', 1)
    ->get();
```

Phương thức `withTrashed` cũng có thể được gọi khi xây dựng một truy vấn mối quan hệ:

```
$flight->history()->withTrashed()->get();
```

Chỉ truy xuất các model xóa mềm

Phương thức `onlyTrashed` sẽ chỉ truy xuất các mô hình đã xóa mềm:

```
$flights = Flight::onlyTrashed()
  ->where('airline_id', 1)
  ->get();
```

Lượt bớt model

Đôi khi bạn có thể muốn xóa định kỳ các model không còn cần thiết. Để thực hiện điều này, bạn có thể thêm trait `Illuminate\Database\Eloquent\Prunable` hoặc `Illuminate\Database\Eloquent\MassPrunable` vào các model mà bạn muốn cắt tỉa định kỳ. Sau khi thêm một trong các trait cần thiết vào model, hãy thực thi phương thức `prunable` mà sẽ trả về query builder của Eloquent giúp giải quyết các model không còn cần thiết nữa:

```
<?php

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

class Flight extends Model
{
  use Prunable;

  /**
   * Get the prunable model query.
   *
   * @return \Illuminate\Database\Eloquent\Builder
   */
```

```
  public function prunable()
  {
    return static::where('created_at', '<=', now()->subMonth());
  }
}
```

Khi đánh dấu model với `Prunable`, thì bạn cũng có thể xác định phương thức pruning trên model. Phương thức này sẽ được gọi trước khi model bị xóa. Phương thức này có thể hữu ích để xóa bất kỳ tài nguyên bổ sung nào được liên kết với model, chẳng hạn như các tập tin được lưu trữ, trước khi model bị xóa vĩnh viễn khỏi cơ sở dữ liệu:

```
/**
 * Prepare the model for pruning.
 *
 * @return void
 */
protected function pruning()
{
  //
}
```

Sau khi cấu hình model của mình với prunable, bạn nên lên lịch cho lệnh Artisan `model:prune` trong class `App\Console\Kernel` của ứng dụng. Bạn có thể tự do chọn khoảng thời gian thích hợp mà lệnh này sẽ được chạy:

```
/**
 * Define the application's command schedule.
 *
 * @param  \Illuminate\Console\Scheduling\Schedule  $schedule
 * @return void
 */
protected function schedule(Schedule $schedule)
{
  $schedule->command('model:prune')->daily();
}
```

Đằng sau hậu trường, lệnh `model:prune` sẽ tự động phát hiện các model "Prunable" trong thư mục *app/Models* của ứng dụng của bạn. Nếu model của bạn ở một vị trí khác, bạn có thể sử dụng tùy chọn `--model` để chỉ định tên class model:

```
$schedule->command('model:prune', [
  '--model' => [Address::class, Flight::class],
])->daily();
```

Nếu bạn muốn loại trừ một số model nào đó trong khi cắt giảm model, thì bạn có thể sử dụng tùy chọn `--except`:

```
$schedule->command('model:prune', [
  '--except' => [Address::class, Flight::class],
])->daily();
```

Bạn có thể kiểm tra truy vấn `prunable` của mình bằng cách thực hiện lệnh `model:prune` với tùy chọn `--pretend`. Khi giả vờ, lệnh `model:prune` sẽ chỉ báo cáo có bao nhiêu record được cắt giảm nếu lệnh thực sự chạy:

```
php artisan model:prune --pretend
```

> **Chú ý:** Các model xóa mềm sẽ bị xóa vĩnh viễn (`forceDelete`) nếu chúng khớp với truy vấn có thể cắt giảm.

Cắt bỏ hàng loạt

Khi các model được đánh dấu bằng trait `Illuminate\Database\Eloquent\MassPrunable`, các model sẽ bị xóa khỏi cơ sở dữ liệu bằng cách sử dụng các truy vấn xóa hàng loạt. Do đó, phương thức `pruning` sẽ không được gọi, cũng như các event model khác `deleting` và `deleted` sẽ không được gửi đi. Điều này là do các model không bao giờ thực sự được truy xuất trước khi xóa, do đó làm cho quá trình cắt giảm hiệu quả hơn nhiều:

```
<?php

namespace App\Models;
```

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

class Flight extends Model
{
  use MassPrunable;

  /**
   * Get the prunable model query.
   *
   * @return \Illuminate\Database\Eloquent\Builder
   */
  public function prunable()
  {
    return static::where('created_at', '<=', now()->subMonth());
  }
}
```

Nhân rộng model

Bạn có thể tạo một bản sao chưa lưu của một đối tượng model hiện có bằng cách sử dụng phương thức replicate. Phương thức này đặc biệt hữu ích khi bạn có các đối tượng model chia sẻ nhiều thuộc tính giống nhau:

```php
use App\Models\Address;

$shipping = Address::create([
  'type' => 'shipping',
  'line_1' => '123 Example Street',
  'city' => 'Victorville',
  'state' => 'CA',
  'postcode' => '90001',
]);

$billing = $shipping->replicate()->fill([
  'type' => 'billing'
```

```
]);

$billing->save();
```

Để loại trừ một hoặc nhiều thuộc tính được sao chép sang model mới, bạn có thể truyền một mảng cho phương thức `replicate`:

```
$flight = Flight::create([
  'destination' => 'LAX',
  'origin' => 'LHR',
  'last_flown' => '2020-03-04 11:00:00',
  'last_pilot_id' => 747,
]);

$flight = $flight->replicate([
  'last_flown',
  'last_pilot_id'
]);
```

Phạm vi query

Phạm vi global - tổng thể

Phạm vi tổng thể cho phép bạn thêm các ràng buộc vào tất cả các truy vấn cho một model nhất định. Chức năng xóa mềm của chính Laravel sẽ sử dụng phạm vi tổng thể để truy xuất các mô hình "không bị xóa" từ cơ sở dữ liệu. Viết ra phạm vi tổng thể của riêng bạn có thể tạo ra một thế thuận tiện, dễ dàng để đảm bảo mọi truy vấn cho một model nào đó đều nhận được những ràng buộc nhất định.

Viết ra phạm vị tổng thể

Viết ra phạm vi tổng thể rất đơn giản. Đầu tiên, xác định một class thực thi interface `Illuminate\Database\Eloquent\Scope`. Laravel không có một vị trí áp đặt nào để bạn đặt các class phạm vi, vì vậy bạn có thể tự do đặt class này trong bất kỳ thư mục nào bạn muốn.

Interface `Scope` yêu cầu bạn thực thi một phương thức: `apply`. Phương thức này có thể

thêm vào các ràng buộc `where` hoặc các loại mệnh đề khác trong truy vấn nếu cần:

```php
<?php

namespace App\Scopes;
use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Scope;

class AncientScope implements Scope
{
  /**
   * Apply the scope to a given Eloquent query builder.
   *
   * @param  \Illuminate\Database\Eloquent\Builder  $builder
   * @param  \Illuminate\Database\Eloquent\Model  $model
   * @return void
   */
  public function apply(Builder $builder, Model $model)
  {
    $builder->where('created_at', '<', now()->subYears(2000));
  }
}
```

> Nếu phạm vi tổng thể của bạn đang thêm các cột vào mệnh đề select của truy vấn, bạn nên sử dụng phương thức `addSelect` thay vì `select`. Điều này sẽ ngăn chặn việc vô tình thay thế mệnh đề lựa chọn hiện có của truy vấn.

Ứng dụng các phạm vi tổng thể

Để gán phạm vi tổng thể cho một model, bạn nên ghi đè phương thức booted của model và gọi phương thức `addGlobalScope` của model. Phương thức `addGlobalScope` chấp nhận một trường hợp phạm vi của bạn làm đối số duy nhất của nó:

```php
<?php

namespace App\Models;
```

```php
use App\Scopes\AncientScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
  /**
   * The "booted" method of the model.
   *
   * @return void
   */
  protected static function booted()
  {
    static::addGlobalScope(new AncientScope);
  }
}
```

Sau khi thêm phạm vi trong ví dụ trên vào model `App\Models\User`, một lệnh gọi đến phương thức `User::all()` sẽ thực thi truy vấn SQL sau:

```sql
select * from `users` where `created_at` < 0021-02-18 00:00:00
```

Phạm vi tổng thể nặc danh

Eloquent cũng cho phép bạn xác định phạm vi tổng thể bằng cách sử dụng các hàm xử lý, điều này đặc biệt hữu ích cho các phạm vi đơn giản nào đó trong việc chứng minh một class riêng lẻ nào đó không giành riêng cho chúng. Khi xác định phạm vi tổng thể bằng cách sử dụng hàm xử lý, thì bạn nên cung cấp tên phạm vi mà bạn chọn làm đối số đầu tiên cho phương thức `addGlobalScope`:

```php
<?php

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

```php
class User extends Model
{
  /**
   * The "booted" method of the model.
   *
   * @return void
   */
  protected static function booted()
  {
    static::addGlobalScope('ancient', function (Builder $builder) {
      $builder->where('created_at', '<', now()->subYears(2000));
    });
  }
}
```

Gở bỏ phạm vi tổng thể

Nếu bạn muốn xóa phạm vi tổng thể cho một truy vấn nhất định, bạn có thể sử dụng phương thức `withoutGlobalScope`. Phương thức này chấp nhận tên class của phạm vi tổng thể làm đối số duy nhất của nó:

```php
User::withoutGlobalScope(AncientScope::class)->get();
```

Hoặc, nếu bạn đã xác định phạm vi tổng thể bằng cách sử dụng hàm xử lý, bạn nên chuyển tên chuỗi câu mà bạn đã gán cho phạm vi tổng thể:

```php
User::withoutGlobalScope('ancient')->get();
```

Nếu bạn muốn xóa một vài hoặc thậm chí tất cả phạm vi tổng thể của truy vấn, bạn có thể sử dụng phương thức `withoutGlobalScope`:

```php
// Remove all of the global scopes...
User::withoutGlobalScopes()->get();

// Remove some of the global scopes...
User::withoutGlobalScopes([
```

```
    FirstScope::class, SecondScope::class
])->get();
```

Phạm vi cục bộ

Phạm vi cục bộ cho phép bạn xác định các bộ ràng buộc truy vấn phổ biến mà bạn có thể dễ dàng sử dụng lại trong toàn bộ ứng dụng của mình. Ví dụ: bạn có thể cần thường xuyên truy xuất tất cả người dùng được coi là "phổ biến". Để xác định phạm vi, hãy thiết lập một tiền tố cho một phương thức Eloquent nào đó với `scope`.

Phạm vi phải luôn trả về cùng một đối tượng query builder hoặc `void`:

```
<?php

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

class User extends Model
{
    /**
     * Scope a query to only include popular users.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $query
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function scopePopular($query)
    {
        return $query->where('votes', '>', 100);
    }

    /**
     * Scope a query to only include active users.
     *
     * @param  \Illuminate\Database\Eloquent\Builder  $query
     * @return void
     */
    public function scopeActive($query)
```

```
  {
    $query->where('active', 1);
  }
}
```

Vận dụng một phạm vị cục bộ

Khi phạm vi đã được xác định, bạn có thể gọi các phương thức phạm vi khi truy vấn model. Tuy nhiên, bạn không nên đưa vào tiền tố `scope` khi gọi phương thức. Bạn thậm chí có thể xâu chuỗi các cuộc gọi đến các phạm vi khác nhau:

```
use App\Models\User;

$users = User::popular()->active()->orderBy('created_at')->get();
```

Việc kết hợp nhiều phạm vi model Eloquent thông qua toán tử truy vấn `or` có thể yêu cầu sử dụng các hàm xử lý để đạt được nhóm logic chính xác:

```
$users = User::popular()->orWhere(function (Builder $query) {
  $query->active();
})->get();
```

Tuy nhiên, vì điều này có thể phức tạp, Laravel cung cấp một phương pháp "bậc cao" `orWhere` cho phép bạn kết nối các phạm vi được xâu chuỗi với nhau một cách trôi chảy mà không cần sử dụng các hàm xử lý:

```
$users = App\Models\User::popular()->orWhere->active()->get();
```

Các phạm vi động

Đôi khi bạn có thể muốn xác định một phạm vi chấp nhận các tham số. Để bắt đầu, chỉ cần thêm các tham số bổ sung của bạn vào hình thức của phương thức phạm vi của bạn. Tham số phạm vi phải được xác định sau tham số `$query`:

```
<?php
```

```php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
  /**
   * Scope a query to only include users of a given type.
   *
   * @param  \Illuminate\Database\Eloquent\Builder  $query
   * @param  mixed  $type
   * @return \Illuminate\Database\Eloquent\Builder
   */
  public function scopeOfType($query, $type)
  {
    return $query->where('type', $type);
  }
}
```

Khi các đối số mong đợi đã được thêm vào hình thức của phương thức phạm vi của bạn, bạn có thể truyền các đối số khi gọi phạm vi:

```php
$users = User::ofType('admin')->get();
```

So sánh các model

Đôi khi bạn có thể cần phải xác định xem hai model có "giống nhau" hay không. Phương thức `is` và `isNot` có thể được sử dụng để nhanh chóng xác minh hai model có cùng khóa chính, bảng và kết nối cơ sở dữ liệu hay không:

```php
if ($post->is($anotherPost)) {
  //
}

if ($post->isNot($anotherPost)) {
```

```
    //
}
```

Các phương thức `is` và `isNot` cũng khả dụng khi sử dụng các mối quan hệ `belongsTo`, `hasOne`, `morphTo` và `morphOne`. Phương thức này đặc biệt hữu ích khi bạn muốn so sánh một model có liên quan mà không đưa ra một truy vấn để truy xuất model đó:

```
if ($post->author()->is($user)) {
    //
}
```

Các Event

> Bạn muốn phát trực tiếp các sự kiện Eloquent tới ứng dụng phía máy khách của mình? Kiểm tra phát sóng sự kiện mô hình của Laravel.

Các model Eloquent gửi đi một số event, cho phép bạn nắm bắt những khoảnh khắc sau trong vòng đời của model: `retrieved`, `creating`, `created`, `updating`, `updated`, `saving`, `saved`, `deleting`, `deleted`, `restoring`, `restored` và `replicating`.

Sự kiện `retrieved` sẽ gửi đi khi một model hiện có được truy xuất từ cơ sở dữ liệu. Khi một model mới được lưu lần đầu tiên, các event đã tạo và đã tạo sẽ gửi đi. Các event `updating`/`updated` sẽ gửi đi khi một model hiện có được sửa đổi và phương thức `save` được gọi. Các sự kiện `saving`/`saved` sẽ gửi đi khi một model được tạo hoặc cập nhật - ngay cả khi các thuộc tính của model không bị thay đổi. Tên event kết thúc bằng `-ing` được gửi đi trước khi bất kỳ thay đổi nào đối với model được duy trì, trong khi các event kết thúc bằng `-ed` được gửi đi sau khi các thay đổi đối với model được duy trì.

Để bắt đầu lắng nghe các model event, hãy xác định thuộc tính `$dispatchesEvents` trên model Eloquent của bạn. Thuộc tính này sẽ bố trí các điểm khác nhau trong vòng đời của model Eloquent cho các class event của riêng bạn. Mỗi class event của model sẽ nhận được một đối tượng của model bị ảnh hưởng thông qua constructor của nó:

```
<?php

namespace App\Models;
use App\Events\UserDeleted;
use App\Events\UserSaved;
```

```php
use Illuminate\Foundation\Auth\User as Authenticatable;

class User extends Authenticatable
{
  use Notifiable;

  /**
   * The event map for the model.
   *
   * @var array
   */
  protected $dispatchesEvents = [
    'saved' => UserSaved::class,
    'deleted' => UserDeleted::class,
  ];
}
```

Sau khi xác định và bố trí các sự kiện Eloquent của mình, bạn có thể sử dụng chương trình theo dõi event để xử lý các event.

> **Chú ý:** Khi đưa ra vấn đề cập nhật hàng loạt hoặc truy vấn xóa qua Eloquent, các event của model như `saved`, `updated`, `deleting` và `deleted` sẽ không được gửi đi cho các model bị ảnh hưởng. Điều này là do các model không bao giờ thực sự được truy xuất khi thực hiện cập nhật hoặc xóa hàng loạt.

Sử dụng các hàm xử lý

Thay vì sử dụng các class event tự tạo, bạn có thể đăng ký các hàm xử lý sẽ thực thi khi các event của model khác nhau được gửi đi. Thông thường, bạn nên đăng ký các hàm này trong phương thức `booted` của model của bạn:

```php
<?php

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

class User extends Model
```

```
{
  /**
   * The "booted" method of the model.
   *
   * @return void
   */
  protected static function booted()
  {
    static::created(function ($user) {
      //
    });
  }
}
```

Nếu cần, bạn có thể sử dụng *chương trình theo dõi event ẩn danh có thể xếp hàng đợi* khi đăng ký các event của model. Điều này sẽ hướng dẫn Laravel thực thi chương trình xử lý event của model trong môi trường nền bằng cách sử dụng tính năng queue trong ứng dụng của bạn:

```
use function Illuminate\Events\queueable;

static::created(queueable(function ($user) {
  //
}));
```

Các chương trình giám sát

Tạo chương trình giám sát

Nếu bạn đang nghe nhiều event trên một model nhất định, bạn có thể sử dụng các chương trình giám sát để nhóm tất cả những chương trình theo dõi của bạn thành một class duy nhất. Các class chương trình giám sát có tên phương thức phản ánh các Eloquent event mà bạn muốn theo dõi. Mỗi phương thức này nhận model bị ảnh hưởng làm đối số duy nhất của chúng. Lệnh Artisan `make:observer` là cách dễ nhất để tạo một class chương trình giám sát mới:

```
php artisan make:observer UserObserver --model=User
```

Lệnh này sẽ đặt chương trình giám sát mới trong thư mục `App/Observers` của bạn. Nếu thư mục này không tồn tại, lệnh Artisan sẽ tạo nó cho bạn. Chương trình giám sát mới của bạn sẽ trông giống như sau:

```php
<?php

namespace App\Observers;

use App\Models\User;

class UserObserver
{
  /**
   * Handle the User "created" event.
   *
   * @param  \App\Models\User  $user
   * @return void
   */
  public function created(User $user)
  {
    //
  }

  /**
   * Handle the User "updated" event.
   *
   * @param  \App\Models\User  $user
   * @return void
   */
  public function updated(User $user)
  {
    //
  }

  /**
```

```
     * Handle the User "deleted" event.
     *
     * @param  \App\Models\User  $user
     * @return void
     */
    public function deleted(User $user)
    {
        //
    }

    /**
     * Handle the User "forceDeleted" event.
     *
     * @param  \App\Models\User  $user
     * @return void
     */
    public function forceDeleted(User $user)
    {
        //
    }
}
```

Để đăng ký một chương trình giám sát, bạn cần gọi phương thức `observe` trên model mà bạn muốn quan sát. Bạn có thể đăng ký giám sát trong phương thức `boot` của nhà cung cấp dịch vụ `App\Providers\EventServiceProvider` của ứng dụng:

```
use App\Models\User;
use App\Observers\UserObserver;

/**
 * Register any events for your application.
 *
 * @return void
 */
public function boot()
{
    User::observe(UserObserver::class);
```

```
}
```

> Có những sự kiện bổ sung mà người quan sát có thể theo dõi, chẳng hạn như `saving` và `retrieved`. Những sự kiện này được mô tả trong tài liệu event.

Các giao dịch transaction

Khi các model đang được tạo trong một giao dịch transaction cơ sở dữ liệu, bạn có thể muốn hướng dẫn một chương trình giám sát chỉ thực thi các hàm xử lý sự kiện của nó sau khi giao dịch transaction cơ sở dữ liệu được chốt. Bạn có thể thực hiện điều này bằng cách xác định thuộc tính `$afterCommit` trên chương trình giám sát. Nếu một giao dịch transaction cơ sở dữ liệu không được tiến hành, các hàm xử lý sự kiện sẽ thực thi ngay lập tức:

```php
<?php

namespace App\Observers;
use App\Models\User;

class UserObserver
{
  /**
   * Handle events after all transactions are committed.
   *
   * @var bool
   */
  public $afterCommit = true;

  /**
   * Handle the User "created" event.
   *
   * @param  \App\Models\User  $user
   * @return void
   */
  public function created(User $user)
  {
    //
```

```
    }
}
```

Phớt lờ các event

Đôi khi bạn có thể cần tạm thời "câm lạng" tất cả các event do một model kích hoạt. Bạn có thể đạt được điều này bằng cách sử dụng phương thức `withoutEvents`. Phương thức `withoutEvents` chấp nhận một hàm xử lý làm đối số duy nhất của nó. Bất kỳ mã nào được thực thi trong hàm này sẽ không gửi các model event và bất kỳ giá trị nào được trả về bởi hàm xử lý sẽ được trả về bởi phương thức `withoutEvents`:

```
use App\Models\User;

$user = User::withoutEvents(function () use () {
    User::findOrFail(1)->delete();

    return User::find(2);
});
```

Lưu model đơn mà không cần các event

Đôi khi bạn có thể muốn "lưu" một model nhất định mà không cần cử đi bất kỳ event nào. Bạn có thể thực hiện điều này bằng phương thức `saveQuietly`:

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

$user->name = 'Victoria Faith';

$user->saveQuietly();
```