NGUYỄN THẾ HUY

ELOQUENT

CÓ THỂ BẠN CHƯA BIẾT

NHỮNG TIPS LÀM VIỆC CÙNG
LARAVEL ELOQUENT

HUYNT57@GMAIL.COM
HTTPS://HUYNT.DEV

Lời mở đầu

Chào bạn, và cám ơn bạn đã đọc ebook này của mình.
Mình là Huy, hiện tại mình là Technical Leader tại Công ty Cổ phần Giao hàng Tiết kiệm. Ngoài ra, mình còn là Admin của Group Laravel Việt Nam, một cộng đồng Laravel lớn trên Facebook với gần 30.000 thành viên.
Ebook này của mình nhằm cùng cấp cho các bạn một số “tips” khi làm việc cùng Laravel Eloquent. Ebook không tập trung vào khái niệm “Eloquent là gì ?”, hay làm sao để bắt đầu với Eloquent. Ebook sẽ phù hợp với các bạn đã có kinh nghiệm cơ bản, mong muốn học hỏi thêm một số tính năng thú vị của Eloquent mà chưa thực sự phổ biến.
Dù bạn là người mới bắt đầu hay đã có kinh nghiệm, mình tin những kiến thức trong ebook này sẽ giúp bạn rất nhiều trong việc chuẩn bị nền tảng kiến thức với Laravel Eloquent, một trong những ORM phổ biến và mạnh mẽ nhất hiện nay trong PHP.
Chúc bạn có những giây phút thú vị, bổ ích khi đọc ebook.

Các tính năng Eloquent thú vị

Section này mình xin trình bày về một số tính năng hay của Eloquent nhưng không quá phổ biến trong các dự án thực tế. Mục đích của section giúp các bạn tìm hiểu về các tính năng này, và tận dụng tối đa sức mạnh mà Eloquent cung cấp cho chúng ta. Các tính năng này được chọn lọc theo quan điểm cá nhân của mình, có thể không hoàn toàn đầy đủ. Mời các bạn tham khảo dưới đây:

Default Attribute Values

Trong Laravel, bạn có thể định nghĩa giá trị mặc định cho các thuộc tính của model bằng cách sử dụng thuộc tính $attributes trong model đó. Điều này rất hữu ích khi bạn muốn chắc chắn rằng một thuộc tính nào đó sẽ luôn có giá trị mặc định nếu nó không được set trước đó. Ví dụ, khi mình khởi tạo một instance của Model, mặc định các giá trị thuộc tính Model sẽ là null. **Default Attribute Values** sẽ giải quyết vấn đề này.
Ví dụ:

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

class User extends Model
{
    // Các thuộc tính có giá trị mặc định
    protected $attributes = [
        'status' => 'active',  // Giá trị mặc định cho 'status' là 'active'
        'role' => 'user',      // Giá trị mặc định cho 'role' là 'user'
    ];
}
```

Với Laravel 9 trở đi, bạn có thể viết bằng cách sử dụng Enum của PHP:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use App\Enums\UserStatus;
use App\Enums\UserRole;

class User extends Model
{
    // Các thuộc tính có giá trị mặc định
    protected $attributes = [
        'status' => UserStatus::Active,  // Giá trị mặc định là 'active'
        'role' => UserRole::User,        // Giá trị mặc định là 'user'
    ];
}
```

Khi bạn tạo một instance mới của model User mà không cung cấp giá trị cho thuộc tính status và role, chúng sẽ tự động được gán các giá trị mặc định đã định nghĩa trong $attributes.

```php
// Tạo một người dùng mới mà không cung cấp giá trị cho 'status' và 'role'
$user = new User();
```

```
// Kiểm tra các thuộc tính
echo $user->status; // Kết quả: 'active'
echo $user->role;   // Kết quả: 'user'
```

Bạn hoàn có thể Override giá trị mặc định:

```
// Ghi đè giá trị mặc định
$user = new User(['status' => 'inactive', 'role' => 'admin']);

echo $user->status; // Kết quả: 'inactive'
echo $user->role;   // Kết quả: 'admin'
```

Với **Default Attribute Values**, bạn có thể tránh được những bug liên quan đến null value, cũng như không cần lặp đi lặp lại các đoạn code check giá trị trong code nữa. Tuy nhiên, cũng hết sức cẩn trọng nhé, vì giá trị mặc định đôi khi cũng khá là “magic” □.

Save Quietly

Thi thoảng, bạn sẽ gặp một project đã hook sẵn các sự kiện như saving, saved, updating, updated …. Tuy nhiên, đôi khi bạn muốn save model mà không kích hoạt các sự kiện này. Để thực hiện điều đó, Laravel cung cấp phương thức saveQuietly().

Ví dụ chúng ta có một Model đã được khai báo sẵn các sự kiện như sau:

```
<?php

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

class Post extends Model
```

```
{
    protected $fillable = ['title', 'content'];

    // Sự kiện 'saving' sẽ được kích hoạt khi sử dụng save()
    protected static function booted()
    {
        static::saving(function ($post) {
            // Logic sự kiện 'saving'
            \Log::info('Post is being saved: ' . $post->title);
        });

        static::saved(function ($post) {
            // Logic sự kiện 'saved'
            \Log::info('Post has been saved: ' . $post->title);
        });
    }
}
```

Sử dụng saveQuietly(), các sự kiện saving, saved sẽ không được kích hoạt:

```
$post = Post::find(1);
$post->title = 'Another Updated Title';

// Sử dụng saveQuietly(), các sự kiện sẽ không được kích hoạt
$post->saveQuietly();
```

Boot Eloquent Traits

Trong Laravel, bạn có thể sử dụng Eloquent Traits để chia sẻ logic chung giữa các model. Khi sử dụng traits với các model, đôi khi bạn cần sử dụng các hook để thực hiện logic khi model được khởi tạo hoặc trong các sự kiện như tạo, cập nhật, xóa. Để thực

hiện điều này, bạn có thể sử dụng phương thức boot cho traits, gọi là boot method của trait.

Laravel hỗ trợ một cách cụ thể để thêm logic khởi tạo (booting) cho các traits bằng cách định nghĩa một phương thức với tên bắt đầu bằng boot và kết thúc bằng tên của trait.

Cách hoạt động:

Khi sử dụng một trait trong một model Eloquent, nếu trait có phương thức boot[TênTrait](), Laravel sẽ tự động gọi nó khi model khởi tạo.

Cấu trúc của boot trong trait:

```php
<?php

trait ExampleTrait
{
    public static function bootExampleTrait()
    {
        // Logic để chạy khi model sử dụng trait này được khởi tạo
    }
}
```

Giả sử bạn có một trait để ghi log mỗi khi một model được tạo hoặc cập nhật.

Tạo Trait LogsActivity

```php
<?php

namespace App\Traits;

use Illuminate\Support\Facades\Log;

trait LogsActivity
{
    // Boot method sẽ được gọi khi model khởi tạo
    public static function bootLogsActivity()
```

```
    {
        static::creating(function ($model) {
            Log::info('Creating event fired for ' .
get_class($model));
        });

        static::updating(function ($model) {
            Log::info('Updating event fired for ' .
get_class($model));
        });
    }
}
```

Sử dụng Trait trong Model Post

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use App\Traits\LogsActivity;

class Post extends Model
{
    use LogsActivity;

    protected $fillable = ['title', 'content'];
}
```

Khi Model Post được sử dụng:

```php
// Khi tạo mới một bài viết, sẽ ghi log về sự kiện 'creating'
$post = new Post();
$post->title = 'New Post';
$post->content = 'This is the content';
$post->save(); // Sự kiện 'creating' được kích hoạt và ghi log

// Khi cập nhật bài viết, sẽ ghi log về sự kiện 'updating'
$post->title = 'Updated Post';
$post->save(); // Sự kiện 'updating' được kích hoạt và ghi log
```

Khi bạn chạy đoạn mã trên, bạn sẽ thấy các dòng log sau:

```
[INFO] Creating event fired for App\Models\Post
[INFO] Updating event fired for App\Models\Post
```

Khi nào nên sử dụng boot trong trait:

- Khi bạn muốn chia sẻ logic khởi tạo giữa nhiều model.
- Khi bạn muốn thêm các hook để xử lý các sự kiện như creating, updating, deleting trong nhiều model mà không cần viết đi viết lại trong các Model.
- Đặc biệt hữu ích cho các tình huống như ghi log, cập nhật thời gian, cập nhật giá trị các thuộc tính, hay kiểm tra dữ liệu trước khi lưu.

Upsert

Phương thức upsert() trong Laravel được sử dụng để thực hiện cả việc chèn nhiều bản ghi mới và cập nhật nếu các bản ghi đã tồn tại, mà không cần phải kiểm tra thủ công xem bản ghi đã tồn tại hay chưa.

Cú pháp upsert() với Eloquent Model

```
Model::upsert(
    $values,        // Mảng các bản ghi cần chèn hoặc cập nhật
    $uniqueBy,      // Cột hoặc các cột để xác định bản ghi duy nhất
    $update         // Cột hoặc các cột cần cập nhật nếu bản ghi đã
tồn tại
```

```
);
```

Mô tả chi tiết các tham số:

- $values: Mảng các bản ghi mà bạn muốn thêm hoặc cập nhật. Mỗi bản ghi phải là một array kết hợp với các cột và giá trị tương ứng.
- $uniqueBy: Một hoặc nhiều cột để xác định bản ghi duy nhất. Nếu giá trị trong những cột này trùng với một bản ghi đã tồn tại, hệ thống sẽ thực hiện cập nhật thay vì chèn bản ghi mới. Chú ý, các cột này cần phải là primaryKey hoặc được đánh index “unique” (Mình có giải thích ở đoạn cuối section).
- $update: Danh sách các cột cần cập nhật nếu bản ghi đã tồn tại.

Upsert với nhiều bản ghi trên Eloquent Model:
Giả sử bạn có model Product với các cột id, name, và price, và bạn muốn thực hiện upsert với các sản phẩm dựa trên id.

```php
use App\Models\Product;

// Dữ liệu sản phẩm
$products = [
    ['id' => 1, 'name' => 'Laptop', 'price' => 1500],
    ['id' => 2, 'name' => 'Phone', 'price' => 800],
    ['id' => 3, 'name' => 'Tablet', 'price' => 600],
];

// Thực hiện upsert với Eloquent
Product::upsert($products, ['id'], ['name', 'price']);
```

- Nếu sản phẩm với id = 1 hoặc id = 2 đã tồn tại, các cột name và price sẽ được cập nhật với giá trị mới.
- Nếu không tồn tại sản phẩm nào với id = 3, một bản ghi mới sẽ được chèn vào bảng.

Upsert với tìm kiếm theo nhiều cột

Trong trường hợp bạn có bảng users với các cột email, role, và bạn muốn thực hiện upsert dựa trên tổ hợp của email và role để xác định bản ghi duy nhất.

```
use App\Models\User;

// Dữ liệu người dùng
$users = [
    ['email' => 'john@example.com', 'role' => 'admin', 'name' =>
'John Doe', 'status' => 'active'],
    ['email' => 'jane@example.com', 'role' => 'user', 'name' => 'Jane
Doe', 'status' => 'inactive'],
];

// Thực hiện upsert với Eloquent
User::upsert($users, ['email', 'role'], ['name', 'status']);
```

Nếu người dùng với email = john@example.com và role = admin đã tồn tại, các cột name và status sẽ được cập nhật.
Nếu người dùng với email = jane@example.com và role = user không tồn tại, một bản ghi mới sẽ được thêm vào bảng.
Với upsert(), Laravel sử dụng query **ON DUPLICATE UPDATE**, từ đó bạn chỉ tốn một query duy nhất để thao tác nhiều bản ghi, giúp nâng performance. Ví dụ truy vấn sẽ như sau:

```
INSERT INTO `products` (`id`, `name`, `price`)
VALUES (1, 'Laptop', 1500), (2, 'Phone', 800)
ON DUPLICATE KEY UPDATE `name` = VALUES(`name`), `price` =
VALUES(`price`);
```

Vì sử dụng **ON DUPLICATE KEY**, có hai lưu ý **cực kỳ quan trọng** khi bạn sử dụng upsert:

- Các cột trong tham số thứ hai của upsert buộc phải **là primary key hoặc được đánh index "unique"**. Nếu các cột này không có các index trên, upsert sẽ không hoạt động đúng.
- Vì tác động nhiều bản ghi cùng lúc, upsert sẽ không kích hoạt bất cứ sự kiện nào của Eloquent Model.

Appends

Trong Laravel, thuộc tính $appends của Eloquent Model cho phép bạn thêm các thuộc tính (fields) tùy chỉnh vào kết quả JSON hoặc mảng của model. Các thuộc tính này không có trong cơ sở dữ liệu, nhưng có thể được tính toán hoặc tạo dựa trên các thuộc tính khác của model.

Cách sử dụng $appends

- $appends là một mảng các thuộc tính hay accessors mà bạn muốn thêm vào kết quả khi model được chuyển đổi sang JSON hoặc Array.
- Các thuộc tính được thêm vào qua $appends phải được định nghĩa bằng accessor – tức là một phương thức trong model có tên bắt đầu bằng get và kết thúc bằng Attribute.

Ví dụ sử dụng $appends

Giả sử bạn có một model User với các cột trong cơ sở dữ liệu như first_name và last_name, và bạn muốn thêm một thuộc tính tùy chỉnh full_name kết hợp hai trường này khi trả về JSON hoặc mảng.

Định nghĩa accessor cho full_name

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    protected $fillable = ['first_name', 'last_name'];

    // Thuộc tính 'full_name' sẽ được thêm vào khi chuyển đổi model sang JSON hoặc mảng
    protected $appends = ['full_name'];

    // Accessor để tính toán giá trị của full_name
    public function getFullNameAttribute()
    {
```

```
        return $this->first_name . ' ' . $this->last_name;
    }
}
```

Khi bạn lấy dữ liệu người dùng và chuyển nó sang JSON hoặc Array, thuộc tính full_name sẽ tự động được thêm (append) vào kết quả.

```
// Lấy người dùng từ cơ sở dữ liệu
$user = User::find(1);

// Chuyển model thành JSON
return $user->toJson();

Kết quả trả về sẽ bao gồm cả full_name:
{
    "id": 1,
    "first_name": "John",
    "last_name": "Doe",
    "full_name": "John Doe",
    "created_at": "2024-10-14 12:00:00",
    "updated_at": "2024-10-14 12:00:00"
}
```

Attribute Cast

Trước khi Laravel giới thiệu **Custom Attribute Casts** bằng phương thức Attribute (từ phiên bản Laravel 8.70 trở đi), thông thường bạn sẽ phải viết như thế này khi làm việc với Accessor và Mutator:

```
<?php

namespace App\Models;

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

```
class User extends Model
{
    protected $fillable = ['first_name', 'last_name'];

    // Accessor: Định nghĩa cách lấy thuộc tính full_name
    public function getFullNameAttribute()
    {
        return "{$this->first_name} {$this->last_name}";
    }

    // Mutator: Định nghĩa cách lưu thuộc tính full_name
    public function setFullNameAttribute($value)
    {
        $names = explode(' ', $value);
        $this->attributes['first_name'] = strtoupper($names[0]);
        $this->attributes['last_name'] = strtoupper($names[1]);
    }
}
```

trong đó:

- **Accessor**: Được sử dụng để định nghĩa cách lấy ra một thuộc tính. Các phương thức này phải tuân theo quy ước đặt tên với tiền tố get và hậu tố Attribute.
- **Mutator**: Được sử dụng để định nghĩa cách lưu vào một thuộc tính. Các phương thức này phải tuân theo quy ước đặt tên với tiền tố set và hậu tố Attribute.

Đội ngũ Laravel cho rằng đây là cách viết không thực sự hay, vì nó yêu cầu tới hai phương thức cho một thuộc tính, cũng như convention khá rắc rối. Từ sau Laravel 8, chúng ta đã có Attribute Cast được viết như thế này, cá nhân mình thấy khá đẹp và gọn gàng:

```
<?php

namespace App\Models;

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

```php
use Illuminate\Database\Eloquent\Casts\Attribute;

class User extends Model
{
    protected $fillable = ['first_name', 'last_name'];

    /**
     * Định nghĩa accessor và mutator cho full_name sử dụng Attribute Casts.
     */
    protected function fullName(): Attribute
    {
        return Attribute::make(
            // Accessor: Lấy ra dữ liệu
            get: fn () => "{$this->first_name} {$this->last_name}",
            // Mutator: Xử lý khi lưu vào cơ sở dữ liệu
            set: fn ($value) => [
                'first_name' => strtoupper(explode(' ', $value)[0]),
                'last_name' => strtoupper(explode(' ', $value)[1]),
            ],
        );
    }
}
```

Cách làm này mang lại lợi ích chính:

- Bạn chỉ cần một method cho cả getter / setter
- Vẫn đảm bảo việc custom giá trị ở Mutator và Accessor như phiên bản cũ, nhưng vì tập trung vào một đối tượng Attribute nên bảo trì cũng dễ dàng và rõ ràng hơn, tránh việc phải nhớ convention như cách viết cũ.

Invisible Database Columns

Invisible Columns là một tính năng được giới thiệu từ **MySQL 8.0.23**, cho phép bạn tạo các cột trong bảng mà chúng không được hiển thị trong các truy vấn SELECT *, nhưng vẫn có thể truy vấn cụ thể nếu bạn chỉ định tên cột. Laravel đã support cột dạng này, bạn có thể sử dụng phương thức invisible() trong migration để tạo cột này:

```
public function up()
    {
        Schema::table('users', function (Blueprint $table) {
            // Thêm cột 'password' với thuộc tính invisible
            $table->string('password')->invisible();
        });
    }
```

Khi bạn lấy một user dạng:

```
$user = User::query()->first();
```

thì $user->password sẽ là null, mặc dù nó chạy select *, do cột password là invisible. Nếu cần đến password, bạn sẽ cần chạy:

```
User::query()->select("password")->first();
```

Query Time Casting

Trong Laravel, từ phiên bản 8.40 trở đi, bạn có thể sử dụng phương thức **withCasts()** để **ép kiểu dữ liệu tại thời điểm truy vấn** mà không cần phải định nghĩa $casts trong model. Điều này giúp bạn linh hoạt hơn khi cần ép kiểu tạm thời cho các thuộc tính của model trong một truy vấn cụ thể.

Phương thức withCasts() cho phép bạn định nghĩa cách các thuộc tính sẽ được chuyển đổi chỉ trong một truy vấn duy nhất, mà không cần phải ép kiểu toàn bộ model mỗi khi sử dụng.

Cách sử dụng withCasts()

```
Model::query()->withCasts([
    'field_name' => 'cast_type',
])->get();
```

- field_name: Tên của cột bạn muốn ép kiểu.
- cast_type: Kiểu dữ liệu mà bạn muốn cột đó được ép kiểu, tương tự như $casts trong model.

Ép kiểu boolean với withCasts():

Giả sử bạn có một bảng users với cột is_active và bạn muốn ép kiểu cột này thành boolean tại thời điểm truy vấn:

```
use App\Models\User;

$users = User::query()
    ->withCasts([
        'is_active' => 'boolean',
    ])
    ->get();

return $users;
```

Trong kết quả, cột is_active sẽ được chuyển đổi thành giá trị boolean (true hoặc false), thay vì số 0 hoặc 1 từ cơ sở dữ liệu.

Ép kiểu datetime với withCasts()

Giả sử bạn có một bảng posts với cột created_at, và bạn muốn ép kiểu created_at thành kiểu Carbon chỉ trong một truy vấn nhất định:

```
use App\Models\Post;

$posts = Post::query()
    ->withCasts([
        'created_at' => 'datetime',
    ])
    ->get();

return $posts;
```

Với cách này, cột created_at sẽ được trả về như một instance của Carbon\Carbon, cho phép bạn sử dụng các phương thức ngày giờ mạnh mẽ của Carbon.

Ép kiểu array hoặc json với withCasts()

Nếu bạn có một bảng products với cột attributes lưu trữ dưới dạng JSON, bạn có thể ép kiểu cột này thành mảng hoặc JSON Object khi truy vấn.

```
use App\Models\Product;
$products = Product::query()
    ->withCasts([
        'attributes' => 'array',  // Ép kiểu thành mảng
    ])
    ->get();

return $products;
```

Với cách này, cột attributes sẽ tự động được chuyển đổi thành một mảng khi bạn truy vấn dữ liệu, cho phép bạn thao tác với mảng ngay lập tức.
Kết hợp withCasts() và where hoặc select
Bạn cũng có thể kết hợp withCasts() với các điều kiện trong truy vấn, chẳng hạn như where, select, hoặc bất kỳ phương thức truy vấn nào khác:

Ví dụ:

```
$users = User::query()
    ->where('is_active', 1)
    ->withCasts([
        'is_active' => 'boolean',
    ])
    ->get();
$posts = Post::query()
    ->select('id', 'title', 'created_at')
    ->withCasts([
```

```
        'created_at' => 'datetime',
    ])
    ->get();
```

Lợi ích lớn nhất của Query Time Casting thông qua phương thức withCasts() đó là bạn tự chủ hoàn toàn việc casting dữ liệu theo truy vấn mà không cần set toàn cục vào Model, giúp cho code linh hoạt hơn và ít impact đến code cũ.

Prevent Lazy Loading

Lazy Loading trong Laravel là kỹ thuật mặc định của Eloquent khi lấy dữ liệu từ relationship. Có thể ví dụ khi bạn query lấy danh sách các Posts, Laravel sẽ không query lấy danh sách Comments của các bài Post đó (kể cả khi bạn có khai báo hasMany giữa Post và Comment). Chỉ khi bạn truy cập vào từng đối tượng của Post trong vòng lặp chẳng hạn, và sử dụng đoạn code: `$post->comments;` Laravel mới tiến hành truy vấn tìm kiếm dữ liệu tương ứng.

Layzy Loading cho phép các mối quan hệ (relationships) của một model chỉ được truy xuất khi cần thiết, tức là khi bạn thực sự truy cập vào thuộc tính đó như ví dụ trên. Tuy rằng cách làm trên tiện lợi, nhưng trong các ứng dụng lớn hoặc khi bạn làm việc với một số lượng lớn bản ghi, nó có thể dẫn đến **vấn đề về performance** (N+1 Query). Giải pháp thường thấy là bạn sẽ tiến hành **Eager Load** trước các relation qua cú pháp with (và Laravel sẽ chạy truy vấn để lấy thêm các Comment vào danh sách các bài Post luôn). Tuy nhiên trong quá trình code, rất có thể bạn sẽ quên mất việc phải **Eager Load** dữ liệu trước, dẫn tới app bị chậm. Và thú vị ở chỗ, nhiều khi ứng dụng của bạn sẽ chỉ chậm khi lên môi trường Production (vì khi đó dữ liệu mới lớn □). Thế nên, ngăn chặn **Lazy Loading** sớm từ môi trường Local là khá cần thiết.

Laravel cung cấp một cơ chế để **ngăn chặn việc sử dụng Lazy Loading** bằng phương thức **preventLazyLoading**. Khi sử dụng phương thức này (có thể bật lên từ ServiceProvider), Laravel sẽ throw Exception nếu bạn không Eager Load các relation.

Khuyến cáo quan trọng: Bạn chỉ nên bật nó ở Local hoặc quá trình Dev thôi, đừng bật khi lên Production nhé □.

```
<?php
namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use Illuminate\Database\Eloquent\Model;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        // Ngăn chặn lazy loading toàn cục trong môi trường phát
triển
        Model::preventLazyLoading(! $this->app->isProduction());
    }
}
```

Giả sử bạn có hai model Post và Comment, với quan hệ one-to-many giữa chúng:

```
class Post extends Model
{
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

Nếu bạn lấy một bài viết và truy cập vào các bình luận của nó mà không eager load, bạn sẽ sử dụng lazy loading:

```
$post = Post::find(1);
$comments = $post->comments; // Lazy load - sẽ thực hiện truy vấn chỉ khi bạn truy cập vào comments
```

Khi **preventLazyLoading** được bật, truy vấn trên sẽ gây ra một exception vì comments chưa được eager load. Điều này sẽ giúp bạn nhanh chóng phát hiện các đoạn code quên không Eager Load trong quá trình phát triển:

```
LazyLoadingViolationException: Attempted to lazy load [comments] on model [App\Models\Post] but lazy loading is disabled.
```

Bạn sẽ cần chuyển sang Eager Load như sau để tránh Laravel ném ra Exception:

```
$post = Post::with('comments')->find(1); // Eager load mối quan hệ comments
$comments = $post->comments; // Không có ngoại lệ, vì đã eager load
```

Prevent Lazy Loading cho một Model cụ thể

Nếu bạn chỉ muốn ngăn chặn lazy loading cho một model cụ thể thay vì toàn bộ ứng dụng (như ví dụ trên qua AppServiceProvider), bạn có thể sử dụng phương thức preventLazyLoading() trên model đó:

```
use Illuminate\Database\Eloquent\Model;
class Post extends Model
{
    // Ngăn chặn lazy loading cho model Post
    protected static function booted()
    {
        static::preventLazyLoading();
    }
}
```

Strict Mode (Từ Laravel 9)

Trong Laravel 9, Strict Mode (chế độ kiểm soát nghiêm ngặt) trong Eloquent được giới thiệu với các tính năng kiểm soát chặt chẽ hơn so với preventLazyLoading:

- Prenvet lazy loading: Chặn hoàn toàn lazy loading như preventLazyLoading. Điều này có nghĩa bạn bắt buộc phải Eager Load trước khi sử dụng relation.
- Throw ra exception khi gán giá trị mà không có trong fillable.
- Không cho phép truy cập thuộc tính mà chưa được load từ database: thêm một bước để ngăn chặn N+1 query.
- Chặn truy cập một thuộc tính không tồn tại trên model (Bình thường bạn có thể truy cập thuộc tính bất kỳ và nó sẽ trả ra null)

Bạn có thể bật Strict Mode ngay từ AppServiceProvider cho toàn bộ ứng dụng thông qua static method ShouldBeStrict():

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

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

Lưu ý: Bạn chỉ nên bật ở môi trường không phải Production nhé. Tuyệt đối không bật trên Production để tránh lỗi không đáng có.

Phương thức **shouldBeStrict()** là tập hợp của ba phương thức sau, tương ứng ba chế độ khác nhau:

```
Model::preventLazyLoading();
```

Chặn LazyLoading như phần trên mình đã mô tả.

```
Model::preventSilentlyDiscardingAttributes();
```

Chặn fill thuộc tính không có trong fillable.

```
Model::preventsAccessingMissingAttributes();
```

Chặn truy cập thuộc tính không tồn tại.

Bạn cũng có thể sử dụng các phương thức này theo cơ chế riêng lẻ tương tự **preventLazyLoading()**.

SaveMany

Phương thức saveMany() trong Laravel Eloquent được sử dụng để lưu nhiều model liên quan cùng một lúc. Phương thức này thường được sử dụng khi bạn có một relation hasMany hoặc belongsToMany giữa các model và bạn muốn lưu nhiều bản ghi vào mối quan hệ này cùng một lúc.
Giả sử bạn có hai model: Post và Comment. Một bài viết (Post) có thể có nhiều bình luận (Comment) – đây là một quan hệ hasMany.

```
class Post extends Model
{
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

```php
class Comment extends Model
{
    public function post()
    {
        return $this->belongsTo(Post::class);
    }
}
```

Bạn có thể sử dụng saveMany() để lưu nhiều bình luận cho một bài viết cùng một lúc:

```php
// Lấy một bài viết từ cơ sở dữ liệu
$post = Post::find(1);

// Tạo nhiều bình luận mới
$comment1 = new Comment(['content' => 'Bình luận thứ nhất']);
$comment2 = new Comment(['content' => 'Bình luận thứ hai']);

// Lưu nhiều bình luận cùng một lúc
$post->comments()->saveMany([$comment1, $comment2]);
```

CreateMany

Tương tự như SaveMany, nhưng thay vì các Model thì bạn có thể sử dụng Array:

```php
// Lấy một bài viết từ cơ sở dữ liệu
$post = Post::find(1);

// Tạo và lưu nhiều bình luận với dữ liệu thô
$post->comments()->createMany([
    ['content' => 'Bình luận thứ nhất'],
    ['content' => 'Bình luận thứ hai'],
    ['content' => 'Bình luận thứ ba'],
]);
```

Pessimistic Locking

Pessimistic Locking là một chiến lược quản lý khóa trong cơ sở dữ liệu, được sử dụng để đảm bảo tính nhất quán và ngăn ngừa xung đột khi nhiều giao dịch cố gắng truy cập và thay đổi cùng một tài nguyên. Trong mô hình này, khi một giao dịch đọc hoặc thay đổi một tài nguyên, nó sẽ khóa tài nguyên đó để ngăn các giao dịch khác sửa đổi nó cho đến khi khóa được giải phóng. Eloquent đã wrap lại và cho chúng ta các phương thức khóa rất hiệu quả trên Database dựa trên Pessimistic Locking. Dưới đây là hai phương thức phổ biến nhất: SharedLock() và LockForUpdate().

sharedLock()

sharedLock() sẽ khóa bản ghi ở chế độ **chỉ đọc**, cho phép các tiến trình khác đọc bản ghi nhưng chặn bất kỳ thay đổi nào như cập nhật hoặc xóa cho đến khi transaction của bạn hoàn thành. Điều này hữu ích trong các trường hợp bạn muốn thực hiện các thao tác đọc trên bản ghi nhưng không muốn các tiến trình khác thay đổi dữ liệu trong khi bạn đang thao tác.

```
$post = Post::where('id', 1)->sharedLock()->first();
```

lockForUpdate()

lockForUpdate() sẽ khóa bản ghi để các tiến trình khác không thể cập nhật hay xóa bản ghi đó cho đến khi transaction hiện tại hoàn thành.

```
$account = Account::where('id', 1)->lockForUpdate()->first();
```

Prunable Trait

Trong Laravel, Prunable là một trait được giới thiệu từ Laravel 8.26 nhằm giúp chúng ta có thể dễ dàng xóa các bản ghi cũ, không còn cần thiết hoặc hết hạn khỏi database một cách tự động. Điều này rất hữu ích cho việc **dọn dẹp cơ sở dữ liệu** và giúp giảm tải những bản ghi không còn giá trị cho nghiệp vụ nữa.

Khi bạn sử dụng Prunable trait, bạn có thể xác định logic để chỉ định các bản ghi cần xóa và sau đó Laravel sẽ thực hiện việc xóa các bản ghi đó theo lịch trình mà bạn chỉ định (đăng ký với Laravel Task Scheduler)

Bạn chỉ cần thêm Prunable trait vào model của bạn và định nghĩa phương thức prunable() để chỉ định các bản ghi nào nên được xóa.

Giả sử bạn có một model Post, và bạn muốn xóa các bài viết cũ hơn 30 ngày.

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Prunable;

class Post extends Model
{
    use Prunable;

    /**
     * Định nghĩa các bản ghi sẽ bị xóa.
     *
     * @return \Illuminate\Database\Eloquent\Builder
     */
    public function prunable()
    {
        // Xóa các bài viết cũ hơn 30 ngày
        return static::where('created_at', '<=', now()->subDays(30));
    }
}
```

Trong phương thức prunable(), bạn định nghĩa một truy vấn để lấy các bản ghi cần được xóa. Ở đây, chúng ta đang chọn các bài viết có thời gian tạo ra đã quá 30 ngày.

Laravel cung cấp một Artisan command để xóa các bản ghi dựa trên truy vấn được xác định trong phương thức prunable(). Bạn có thể chạy lệnh này thủ công hoặc thiết lập nó để chạy theo lịch trình bằng Task **Scheduler**.

Lệnh Artisan để chạy prune dữ liệu:

```
php artisan model:prune
```

Để tự động hóa quá trình xóa các bản ghi cũ, bạn có thể thêm lệnh **model:prune** vào Scheduler của Laravel. Ví dụ, bạn có thể lên lịch để xóa các bản ghi cũ mỗi ngày:

```
// app/Console/Kernel.php
protected function schedule(Schedule $schedule)
{
    $schedule->command('model:prune')->daily();
}
```

Bạn cũng có thể chỉ định Model muốn chạy prune:

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

Hoặc loại bỏ một số model khỏi việc prune:

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

Một command nữa khá hữu ích là:

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

Khi chạy lệnh này, Laravel sẽ không thực sự xoá các bản ghi mà chỉ report số bản ghi được xoá, tiện cho bạn kiểm tra tính đúng đắn của dữ liệu.

Tappable Scope

Các bạn chắc chắn không xa lạ gì với khái niệm Scope khi làm việc với Eloquent trong Laravel. Scope khá là tiện lợi, nó giúp mình tái sử dụng các điều kiện linh hoạt mà không cần phải define lại code. Nhưng có một cái khá là không thoải mái khi mình dùng scope, đó là IDE gặp khó khăn khi auto suggest scope, khá là bất tiện. Chưa kể là nhiều khi, scope được viết hết vào một file model, nhiều khi làm file của mình phình to lên, khó kiểm soát.

Chưa kể nữa là một số Model của mình có những scope rất giống nhau, ví dụ scopeIsActive chẳng hạn. Mình muốn chỉ viết một scope và có thể apply cho nhiều model cùng lúc thay vì phải copy các scope giống nhau giữa các model.

Tappable Scope sẽ giúp mình giải quyết điều này. Điều thú vị là, cách viết này không đượcc ghi trong Document, nên sẽ không nhiều người biết tới. Những lợi ích đáng kể của Tappable Scope có thể kể tới như: suggestion tốt hơn, dễ dàng chia nhỏ nghiệp vụ khi cần thiết, đảm bảo tính tái sử dụng cho nhiều Model. Nếu phát triển thêm, chúng ta thấy hao hao nét của Specification Design Pattern trong cách viết này 😀

Ví dụ sau đây sẽ giúp bạn hiểu về Tappable Scope:

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

class IsActive
{
    public function __invoke(Builder $builder): void
    {
        $builder->where('status', 'active');
    }
}

class HasVerifiedEmail
{
    public function __invoke(Builder $builder): void
    {
        $builder->whereNotNull('email_verified_at');
    }
}

// Trong controller
```

```
public function getActiveUsersWithVerifiedEmail()
{
    return User::query()
        ->tap(new IsActive)
        ->tap(new HasVerifiedEmail)
        ->get();
}
```

Với Tappable Scope, mỗi Scope được tách thành một class riêng biệt và kích hoạt qua magic method __invoke. Điều này đem đến một số lợi ích như:

- IDE dễ dàng suggest code
- Chia nhỏ được logic nghiệp vụ các scope riêng biệt thay vì viết chung hết vào model
- Tái sử dụng được scope cho các model khác nhau

Custom Query Builder

Eloquent cho phép bạn định nghĩa các **Custom Query Builder**, với mục đích chia tách các logic, tái sử dụng code tốt hơn. Với các ứng dụng không quá chặt chẽ về nghiệp vụ, mình hay áp dụng Custom Query Builder hơn là sử dụng Repository Pattern để linh hoạt và Laravel-way hơn. **(Nhưng các bạn vẫn có thể sử dụng Repository thoải mái nhé, ý mình không phải là thay thế đâu □)**

Vẫn là ví dụ Post và Comment, giờ mình sẽ viết một Custom Query Builder cho Post:

```
namespace App\QueryBuilders;

use Illuminate\Database\Eloquent\Builder;

class PostQueryBuilder extends Builder
{
    /**
     * Lấy những bài viết có số lượng bình luận lớn hơn số được chỉ định.
```

```
     *
     * @param int $count
     * @return self
     */
    public function hasMoreThanComments(int $count): self
    {
        return $this->has('comments', '>', $count);
    }

    /**
     * Lọc các bài viết được tạo trong khoảng thời gian nhất định.
     *
     * @param string $date
     * @return self
     */
    public function createdAfter(string $date): self
    {
        return $this->where('created_at', '>=', $date);
    }
}
```

- hasMoreThanComments(): Phương thức này sẽ thêm điều kiện để chỉ lấy các bài viết có số lượng bình luận lớn hơn số được chỉ định.
- createdAfter(): Phương thức này sẽ thêm điều kiện để chỉ lấy các bài viết được tạo sau một ngày cụ thể.

Tiếp theo, chúng ta sẽ cập nhật model Post để sử dụng **Custom Query Builder** mà chúng ta vừa tạo.

```
namespace App\Models;

use App\QueryBuilders\PostQueryBuilder;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
```

```
{
    protected $guarded = [];

    /**
     * Override newEloquentBuilder để sử dụng Custom Query Builder
     *
     * @param \Illuminate\Database\Query\Builder $query
     * @return PostQueryBuilder
     */
    public function newEloquentBuilder($query): PostQueryBuilder
    {
        return new PostQueryBuilder($query);
    }

    /**
     * Mối quan hệ với model Comment (Một bài viết có nhiều bình
luận)
     */
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

Sau khi thiết lập Custom Query Builder cho model `Post`, chúng ta có thể sử dụng các phương thức trong Custom Query Builder như hasMoreThanComments() và createdAfter().

Ví dụ bạn có thể lấy các bài viết có nhiều hơn 5 bình luận

```
$posts = Post::hasMoreThanComments(5)->get();
foreach ($posts as $post) {
    echo $post->title . ' có hơn 5 bình luận.';
}
```

Hoặc lấy các bài viết có hơn 5 bình luận và được tạo sau ngày 1 tháng 1 năm 2024

```
$posts = Post::hasMoreThanComments(5)
    ->createdAfter('2023-01-01')
    ->get();

foreach ($posts as $post) {
    echo $post->title . ' có hơn 5 bình luận và được tạo sau 01-01-2024.' . PHP_EOL;
}
```

Custom Eloquent Collection

Mặc định, kết quả truy vấn từ Eloquent sẽ là một instance của **Illuminate\Database\Eloquent\Collection** (nếu kết quả là danh sách). Về cơ bản, Collection của Laravel đã rất mạnh mẽ với hàng chục các helper hỗ trợ làm việc với dữ liệu dạng list. Tuy nhiên, trong nhiều trường hợp, bạn sẽ muốn custom Collection này để thêm thắt các phương thức phù hợp cho dự án, cũng như để tránh lặp đi lặp lại các đoạn code xử lý nữa. Tất nhiên là Laravel cho phép bạn làm điều này, tương tự như Custom Query Builder □.

Cùng mình đi qua một số bước đơn giản để tự tạo một Custom Collection nhé. Đầu tiên, chúng ta sẽ cần kế thừa lại Illuminate\Database\Eloquent\Collection và định nghĩa các phương thức chúng ta mong muốn:

```
namespace App\Collections;

use Illuminate\Database\Eloquent\Collection;

class PostCollection extends Collection
{
    // Phương thức để lọc các bài viết đã publish
    public function published()
    {
        return $this->filter(function ($post) {
            return $post->is_published;
        });
    }
}
```

Tiếp theo, bạn sẽ cần áp dụng Custom Collection cho model bằng cách override phương thức **newCollection()** của model đó.

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;
use App\Collections\PostCollection;

class Post extends Model
{
    // Gán collection mới cho model Post
    public function newCollection(array $models = [])
    {
        return new PostCollection($models);
    }
}
```

Gần đây Laravel 11 đã cung cấp thêm một cách nữa để bạn có thể đăng ký Custom Collection, đó là thông qua Attribute CollectedBy:

```
<?php

namespace App\Models;

use App\Collections\PostCollection;
use Illuminate\Database\Eloquent\Attributes\CollectedBy;
use Illuminate\Database\Eloquent\Model;

#[CollectedBy(PostCollection::class)]
class Post extends Model
{
   // ...
}
```

Trông ngắn gọn hơn nhiều việc phải override lại method newCollection nhỉ □

Tips: Nếu bạn muốn dùng CustomCollection cho nhiều Model, hãy tạo một class cha (có sử dụng Custom Collection), và các model của bạn extends từ đó là được nhé.

Với cách triển khai này, mỗi lần bạn lấy các bản ghi từ model Post, Laravel sẽ trả về một instance của PostCollection thay vì collection mặc định. Từ giờ, bạn có thể gọi phương thức này trong kết quả trả về một cách rất tiện lợi như sau:

```
$posts = Post::all()->published(); // Chỉ lấy các bài viết đã được publish
```

Chúng ta có thể custom nhiều hơn nữa PostCollection:

```
namespace App\Collections;

use Illuminate\Database\Eloquent\Collection;

class PostCollection extends Collection
{
    public function published()
    {
        return $this->filter(function ($post) {
            return $post->is_published;
        });
    }

    public function unpublished()
    {
        return $this->filter(function ($post) {
            return !$post->is_published;
        });
    }

    public function withComments()
    {
        return $this->filter(function ($post) {
            return $post->comments()->count() > 0;
        });
    }
}
```

Và đương nhiên rồi, bạn có thể sử dụng chúng một cách rất linh hoạt:

```
// Lấy tất cả bài viết đã publish
$publishedPosts = Post::all()->published();

// Lấy tất cả bài viết chưa publish
$unpublishedPosts = Post::all()->unpublished();

// Lấy tất cả bài viết có bình luận
$postsWithComments = Post::all()->withComments();
```

Những lợi ích của Custom Query Collection hoàn toàn giống Custom Query Builder, giúp bạn tránh lặp đi lặp lại code, dễ dàng áp dụng cho nhiều model khác nhau, tập trung logic tại một chỗ. Đây là hai tính năng rất hay, đừng bỏ qua nhé ae □.

Làm việc với Relationship

Relation hay quan hệ là cụm tính năng cực kỳ mạnh của Eloquent. Trong thực tế, chúng ta hầu như luôn cần đến các relation này khi xử lý nghiệp vụ. Tuy nhiên, xử lý relation cũng có rất nhiều rủi ro về Performance, hay tính đúng đắn của dữ liệu. Section dưới đây mình trình bày một số tính năng của Relation mà mình cho là thú vị, nhưng còn ít người biết tới. Hy vọng nó có thể giúp bạn tối ưu được công việc dev hàng ngày. Lưu ý, vì là chọn lọc, nên nó sẽ không phải toàn bộ các kiến thức về Relation trong Laravel □

Set Relation

Phương thức **setRelation** trong Laravel Eloquent được sử dụng để thiết lập hoặc cập nhật một relation cho một model. Nó giúp bạn gán kết quả của một mối quan hệ đã được eager loading trước đó hoặc gán bằng tay relation vào model mà không cần query lại vào database. Một use case rất thú vị của phương thức này, đó là khi ứng dụng của bạn xuất hiện Circular Dependency. Hãy tưởng tượng ví dụ đơn giản về Post và Comment. Vì một đối tượng, có nhiều bài viết, nên bạn chắc chắn sẽ có relation HasMany Comments.

Khi xử lý các comments với Post, bạn có thể Eager Loading qua with như thế này:

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

Tuy nhiên, khi bạn duyệt từng comment trong relation và trỏ ngược về post ban đầu, thì bạn lại dính N+1 Query:

```
foreach ($post->comments as $comment) {
      $comment->post; // Laravel sẽ chạy thêm một câu truy vấn để query lại Post
}
```

Điều này khá khó chịu, vì rõ ràng là $post đã được query trước đây. Để giải quyết vấn đề này, chúng ta dùng setRelation để set lại $post vào relation của comment:

```
$post->comments->map(function ($comment) use ($post) {
    // Gán post cho mỗi comment
    return $comment->setRelation('post', $post);
});
```

Sau đoạn code này, mỗi $comment đã có thể truy cập tới post mà không cần thêm truy vấn tới Database rồi.

```
foreach ($post->comments as $comment) {
    echo "Comment: " . $comment->content;
    echo "Belongs to Post: " . $comment->post->title; // Không cần thêm truy vấn
}
```

Trong thực tế, tình huống kiểu như này sẽ xảy ra thường xuyên. Ngoài ra bạn có thể dùng setRelation set thủ công quan hệ cho Model mà không nhất thiết phải eager loading ngay từ query ban đầu. Điều này chắc chắn sẽ giúp code của bạn linh hoạt hơn và tối ưu không nhỏ cho performance.

ChaperOne

ChaperOne là tính năng mới của Laravel kể từ phiên bản 11 (tính năng được thêm vào đâu đó tháng 9/2024). Nôm na, ChaperOne là cách giải quyết đẹp hơn cho tình huống của setRelation, mà bạn không cần loop thủ công từng comment như trên để set Post nữa. Cùng mình tham khảo ví dụ dưới đây nhé:

```php
<?php

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

// Mặc dù bạn đã eager loading với with, nhưng khi bạn truy cập
parent class (Post) từ Comment, bạn sẽ
// vẫn bị N+1 Query (Tức là Laravel query lại DB để ra đối tượng Post
khi chạy $comment->post)
foreach ($posts as $post) {
    foreach ($post->comments as $comment) {
        echo $comment->title;
    }
}

// Khai báo trong model giúp tự động gán parent class cho các comment
// Khi sử dụng comment->post sẽ không bị N+1 query

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

// Hoặc bạn có thể sử dụng phương thức chaperone khi eager loading để
giải quyết
```

```
use App\Models\Post;

$posts = Post::with([
    'comments' => fn ($comments) => $comments->chaperone(),
])->get();
```

Cách viết này rõ ràng là đẹp và Laravel way hơn đúng không □. Nhưng vì chỉ có từ Laravel 11, nên từ các phiên bản thấp, bạn sẽ vẫn cần tới cách làm của setRelation như bên trên.

Where Relation

Giả sử bạn có hai model: Post và Comment, với mối quan hệ một bài viết có nhiều bình luận (hasMany). Bạn có thể sử dụng whereRelation() để lọc các bài viết dựa trên các điều kiện liên quan đến bình luận. Ví dụ, bạn muốn lấy các bài viết được tạo sau một ngày cố định

```
$posts = Post::whereRelation('comments', 'created_at', '>', '2024-01-01')->get();

foreach ($posts as $post) {
    echo $post->title . ' có bình luận được tạo sau 01-01-2024.';
}
```

whereRelation khá tương đồng với whereHas, nhưng cú pháp ngắn gọn, phù hợp với các truy vấn đơn giản hơn, nên mình đưa vào mục đầu tiên này □.

Lazy Eager Loading (load)

Lazy Eager Loading trong Laravel là một khái niệm kết hợp giữa **Lazy Loading** và **Eager Loading**. Nó cho phép bạn **lazy load** relation **sau khi** bạn đã truy vấn model từ cơ sở dữ liệu (tức là ở truy vấn đầu tiên, Laravel sẽ không thực hiện lấy relation model

ngay lập tức như khi bạn thực hiện với phương thức with()), nhưng theo cách **eager load** (tức là query một lần lấy hết các model liên quan luôn như with()). Điều này có thể hữu ích khi bạn chưa biết mình sẽ cần relation nào khi bắt đầu truy vấn, nhưng muốn tối ưu hoá lượng query khi bắt đầu load relation sau đó.

Laravel cung cấp phương thức load() để thực hiện lazy eager loading.
Bạn có thể sử dụng phương thức load() sau khi đã có tập dữ liệu của model để “load” các “relation” tương ứng.
Ví dụ:
Giả sử bạn có model Post và model Comment với quan hệ one-to-many giữa chúng:

```
class Post extends Model
{
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

Nếu bạn đã truy vấn các Post nhưng chưa eager load các bình luận (vì có thể bạn cảm thấy chưa cần tới), bạn có thể sử dụng load() để lazy eager load các bình luận ở phía sau, vẫn đảm bảo rằng mình không vướng vào N+1 Query:

```
// Lấy tất cả các bài viết mà chưa eager load bình luận
$posts = Post::all();

// Sau đó, lazy eager load các bình luận
$posts->load('comments');

// Bây giờ bạn có thể truy cập comments mà không cần thêm truy vấn
nào nữa
foreach ($posts as $post) {
    foreach ($post->comments as $comment) {
        echo $comment->content;
    }
}
```

Trong ví dụ trên:

- Ban đầu bạn lấy tất cả các Post mà không eager load comments.
- Sau đó, khi bạn đã có các Post, bạn sử dụng load() để lazy eager load các bình luận cho tất cả các bài viết trong một truy vấn duy nhất.

Lazy Eager Loading cho nhiều mối quan hệ

Bạn có thể lazy eager load nhiều mối quan hệ cùng lúc bằng cách truyền vào một mảng các relation cho phương thức load():

```
$posts = Post::all();

// Load nhiều mối quan hệ
$posts->load(['comments', 'author', 'tags']);
```

Lazy Eager Loading với điều kiện (Constraints)

Bạn có thể thêm điều kiện khi thực hiện lazy eager load bằng cách sử dụng closure:

```
$posts = Post::all();

// Lazy eager load các bình luận nhưng chỉ lấy những bình luận được
phê duyệt
$posts->load(['comments' => function ($query) {
    $query->where('approved', true);
}]);
```

Trong ví dụ này, chỉ những bình luận đã được phê duyệt (approved = true) sẽ được lazy eager load.

Lazy Eager Loading trên một Model cụ thể

Bạn cũng có thể lazy eager load một quan hệ trên một instance của model thay vì trên một tập hợp các model.

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

// Lazy eager load bình luận cho một post cụ thể
$post->load('comments');

// Bây giờ có thể truy cập vào comments mà không cần thêm truy vấn nào nữa
foreach ($post->comments as $comment) {
    echo $comment->content;
}
```

LoadMissing

loadMissing() là một phương thức trong Laravel Eloquent được sử dụng để eager load realtion chỉ khi relation đó chưa được load. Điều này đặc biệt hữu ích khi bạn đang làm việc với một model hoặc một tập hợp các model, và không chắc liệu các relation đã được eager load hay chưa. Điều này sẽ giúp bạn tránh thực hiện gọi load() lên nhiều lần, gây lãng phí các truy vấn. Trong nhiều trường hợp, khi các mối quan hệ đã được eager load trước đó, sử dụng load() có thể dẫn đến các truy vấn không cần thiết. loadMissing() chỉ thực hiện truy vấn để load các relation chưa được truy vấn trước đó, giúp tối ưu hóa performance.

Giả sử bạn có model Post và Comment với quan hệ one-to-many giữa chúng. Bạn muốn lấy ra các bài viết cùng với các bình luận, nhưng chỉ lazy eager load các bình luận nếu chúng chưa được lấy ra.

```
$posts = Post::all();

// Lazy eager load các bình luận chỉ nếu chúng chưa được load
```

```
$posts->loadMissing('comments');

foreach ($posts as $post) {
    foreach ($post->comments as $comment) {
        echo $comment->content;
    }
}
```

Các tính năng còn lại của loadMissing() tương tự với load().

Push

Thi thoảng bạn sẽ muốn lưu Model và cả Relation đi cùng với nó. Phương thức save() sẽ chỉ lưu model mà thôi. Đây là lúc bạn sẽ cần tới push()
Giả sử bạn có hai model Post và Comment với quan hệ **one-to-many** (một bài viết có nhiều bình luận):

```
class Post extends Model
{
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}

class Comment extends Model
{
    public function post()
    {
        return $this->belongsTo(Post::class);
    }
}
```

Giả sử bạn cần cập nhật một bài viết và đồng thời cập nhật một bình luận của bài viết đó. Thay vì lưu riêng bài viết và bình luận, bạn có thể sử dụng **push()** để lưu cả hai cùng một lúc.

```
// Lấy bài viết và load các bình luận liên quan
$post = Post::with('comments')->find(1);

// Thay đổi title của bài viết
$post->title = 'New Title for the Post';

// Thay đổi nội dung của bình luận đầu tiên
$post->comments[0]->content = 'Updated content for the first
comment';

// Sử dụng push để lưu cả bài viết và bình luận
$post->push();
```

Laravel thậm chí cung cấp cho bạn cả **pushQuietly()** tương tự với **saveQuietly()** nhằm mục đích không kích hoạt bất cứ Event Model nào. Quá ngon □.

WhenLoaded

Phương thức whenLoaded() trong Laravel Eloquent là một cách tiện lợi để kiểm tra xem một mối quan hệ (relationship) đã được load hay chưa, trước khi bạn thao tác hoặc truy cập vào nó. Điều này giúp bạn tránh gặp phải các lỗi khi cố gắng truy cập vào một mối quan hệ chưa được load, cũng như kiểm tra xem mình có đang vướng vào vấn đề N+1 Query hay không:

```
$model->whenLoaded('relationship_name', function ($relationship) {
    // Thao tác với $relationship khi nó đã được load
}, function () {
    // Tùy chọn: hành động khi relationship chưa được load
});
```

Giả sử bạn có model Post với mối quan hệ comments (một bài viết có nhiều bình luận). Bạn muốn thao tác với comments nhưng chỉ khi mối quan hệ này đã được load.

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

```
$post->whenLoaded('comments', function ($comments) {
    foreach ($comments as $comment) {
        echo $comment->content;
    }
});
```

Trong ví dụ trên:

- Nếu comments đã được eager load bằng with('comments'), callback đầu tiên sẽ được thực thi và bạn có thể truy cập vào các bình luận.
- Nếu comments chưa được load, không có hành động nào sẽ được thực hiện.

WhenCounted

Trong Laravel, **whenCounted()** là một phương thức được sử dụng để kiểm tra xem một mối quan hệ (relationship) đã được đếm thông qua withCount() hay chưa. Cách hoạt động của nó cũng tương tự withLoaded(). Sử dụng whenCounted() giúp bạn tránh được vấn đề performance khi relation chưa được count qua withCount(), từ đó giúp bạn ra quyết định dễ dàng hơn.

Giả sử bạn có model Post với mối quan hệ comments (một bài viết có nhiều bình luận), và bạn muốn kiểm tra xem số lượng bình luận đã được đếm hay chưa.

```
// Lấy bài viết cùng với số lượng bình luận
$post = Post::withCount('comments')->find(1);

$post->whenCounted('comments', function ($count) {
    echo "This post has {$count} comments.";
}, function () {
    echo "Comments have not been counted.";
});
```

Trong ví dụ trên:

- withCount('comments') sẽ tính toán và lưu số lượng bình luận vào thuộc tính comments_count.
- whenCounted('comments') sẽ kiểm tra xem withCount() đã được chạy hay chưa. Nếu đã sử dụng wihtCount(), callback đầu tiên sẽ được thực thi. Ngược lại, callback thứ hai sẽ echo ra: “Comments have not been counted.”. Điều này có thể ngăn chặn việc bạn truy cập vào comments_count() (chỉ tồn tại khi bạn sử dụng withCount()).

WithDefault

Trong Laravel Eloquent, phương thức **withDefault()** được sử dụng để gán một giá trị mặc định cho các mối quan hệ belongsTo, hasOne, hasOneThourh hoặc morphOne khi relation đó không tồn tại trong cơ sở dữ liệu. Thay vì trả về null khi relation không được tìm thấy, bạn có thể sử dụng withDefault() để trả về một đối tượng mặc định cho relation này.
withDefault() sẽ hữu dụng khi:

- Khi bạn có relation belongsTo hoặc các relation tương tự mà không phải lúc nào cũng có giá trị trong cơ sở dữ liệu (ví dụ: một bài viết có thể không có tác giả).
- Khi bạn muốn tránh lỗi truy cập thuộc tính từ một giá trị null và thay vào đó trả về một đối tượng mặc định với các thuộc tính đã được định nghĩa.
- Khi bạn muốn trả về giá trị mặc định mà không cần phải kiểm tra thủ công xem relation có tồn tại hay không.

Giả sử bạn có model Post và Author, với mối quan hệ belongsTo giữa chúng (mỗi bài viết thuộc về một tác giả). Một số bài viết có thể không có tác giả trong cơ sở dữ liệu. Bạn có thể sử dụng withDefault() để trả về một đối tượng mặc định nếu không có tác giả nào được liên kết với bài viết.
Model Post với belongsTo và withDefault():

```
class Post extends Model
{
    public function author()
```

```
    {
        return $this->belongsTo(Author::class)->withDefault([
            'name' => 'Guest Author',
        ]);
    }
}
```

Truy vấn sử dụng withDefault():

```
$post = Post::find(1);
// Nếu bài viết không có tác giả, `withDefault()` sẽ trả về một đối tượng Author với giá trị mặc định
echo $post->author->name; // Nếu không có tác giả, sẽ hiển thị "Guest Author"
```

Nếu bài viết không có tác giả (cột author_id trống hoặc không tìm thấy trong bảng authors), phương thức withDefault() sẽ trả về một đối tượng Author với thuộc tính name được gán giá trị "Guest Author".

Hoặc một ví dụ khác:

```
class Post extends Model
{
    public function author()
    {
        return $this->belongsTo(Author::class)->withDefault(function ($author) {
            $author->name = 'Anonymous';
            $author->email = 'no-reply@example.com';
        });
    }
}
```

Khi query:

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

```
// Nếu bài viết không có tác giả, đối tượng Author sẽ được tạo với
name là "Anonymous" và email "no-reply@example.com"
echo $post->author->name;   // "Anonymous"
echo $post->author->email;  // "no-reply@example.com"
```

withDefault() có thể khiến bạn liên tưởng đến **Null Object Design Pattern**, một Pattern giúp bạn không cần các điều kiện check giá trị Null lặp đi lặp lại trong code. Điều này có thể giúp code của bạn an toàn hơn cho các truy vấn sau này.

oneOfMany

Trong các dự án thực tế, có nhiều tình huống bạn chỉ cần lấy một bản ghi duy nhất từ một tập hợp con các bản ghi liên quan. Ví dụ:

- Bạn muốn lấy bài viết mới nhất của một người dùng từ các bài viết của họ.
- Bạn cần lấy đơn hàng gần nhất của một khách hàng.

Thay vì sử dụng quan hệ hasMany và sau đó phải lọc các bản ghi trong ứng dụng, bạn có thể sử dụng quan hệ "one-of-many" để giải quyết vấn đề này trực tiếp từ cơ sở dữ liệu.

Sử dụng ofMany

Phương thức ofMany là điểm chính của quan hệ "one-of-many". Bạn có thể sử dụng nó để lấy bản ghi có giá trị lớn nhất, nhỏ nhất, hoặc bất kỳ điều kiện nào khác từ tập hợp bản ghi liên quan.

Ví dụ: Lấy bài viết mới nhất

Giả sử bạn có bảng posts chứa thông tin về các bài viết của người dùng, và bạn muốn lấy bài viết mới nhất của mỗi người dùng. Đây là cách sử dụng quan hệ "one-of-many":

```
class User extends Model
{
    public function latestPost()
    {
        return $this->hasOne(Post::class)->ofMany('created_at',
'max');
    }
```

```
}
```

Ở đây, ofMany('created_at', 'max') xác định rằng bạn muốn lấy bài viết có created_at lớn nhất, tức là bài viết mới nhất.
Khi gọi relation này, bạn có thể lấy bài viết mới nhất cho một người dùng cụ thể:

```
$user = User::with('latestPost')->find(1);
echo $user->latestPost->title;
```

Ví dụ: Lấy bài viết cũ nhất
Bạn cũng có thể lấy bài viết cũ nhất bằng cách thay đổi điều kiện trong ofMany:

```
class User extends Model
{
    public function oldestPost()
    {
        return $this->hasOne(Post::class)->ofMany('created_at',
'min');
    }
}
```

Trong trường hợp này, ofMany('created_at', 'min') sẽ trả về bài viết có thời gian tạo nhỏ nhất, tức là bài viết cũ nhất.

LatestOfMany() và OldestOfMany()

Laravel đã bổ sung sẵn cho chúng ta latestOfMany và oldestOfMany, giúp đơn giản hóa việc lấy bản ghi mới nhất hoặc cũ nhất từ một quan hệ hasMany hoặc hasOne. Đây là các shortcut cho ofMany mà không cần phải chỉ định cột và điều kiện bằng tay (như ví dụ trên).
Vẫn với ví dụ trên, các bạn có thể viết:

```
class User extends Model
{
    public function latestPost()
    {
        return $this->hasOne(Post::class)->latestOfMany(); // hoặc
```

```
oldestOfMany();
    }
}
```

Mặc định, các phương thức latestOfMany và oldestOfMany trong Laravel sẽ truy vấn bản ghi mới nhất hoặc cũ nhất dựa trên **khóa chính (primary key)** của model liên quan, và ta hiểu rằng khoá chính thì mặc nhiên hỗ trợ sắp xếp được. Câu truy vấn Laravel sẽ tạo ra khi sử dụng relation ofMany có dạng:

```
SELECT * FROM `posts` INNER JOIN ( SELECT MAX(id) AS id FROM posts
GROUP BY posts.user_id ) AS latest_post ON latest_post.id = users.id
```

Laravel cũng warning hiện tại relation này chưa hỗ trợ cột uuid trong Postgres do PostgresSQL chưa support hàm MAX với cột giá trị này. Bạn hãy hết sức chú ý khi sử dụng relation này để tránh những tình huống ngoài ý muốn nhé.

hasManyThrough

Trong Laravel Eloquent, hasManyThrough là một loại quan hệ cho phép bạn định nghĩa một relation gián tiếp thông qua một model trung gian. Trong thực tế, bạn sẽ rất hay gặp các tình huống dạng $parent->child->child. Ví dụ hay gặp nhất thường sẽ là ba bảng, dạng:

- Country (Quốc gia): Bảng quốc gia.
- User (Người dùng): Mỗi quốc gia có nhiều người dùng.
- Post (Bài viết): Mỗi người dùng có nhiều bài viết.

Trong ví dụ này, bạn có thể truy xuất tất cả các bài viết của một quốc gia thông qua người dùng của quốc gia đó. Đây là trường hợp điển hình sử dụng hasManyThrough. Giả sử bạn có ba model: Country, User, và Post. Mối quan hệ giữa chúng là:

- Một quốc gia (Country) có nhiều người dùng (User).
- Một người dùng (User) có nhiều bài viết (Post).

Nếu bạn muốn truy xuất tất cả các bài viết (Post) của một quốc gia, bạn có thể sử dụng hasManyThrough để lấy tất cả các bài viết của một quốc gia thông qua model User. Đầu tiên bạn định nghĩa quan hệ hasManyThrough trong model chính (ví dụ: Country) bằng cách sử dụng phương thức hasManyThrough().

```php
class Country extends Model
{
    public function posts()
    {
        return $this->hasManyThrough(Post::class, User::class);
    }
}
```

Với quan hệ này, bạn có thể truy xuất tất cả các bài viết từ một quốc gia như sau:

```php
$country = Country::find(1);
// Lấy tất cả bài viết của quốc gia thông qua người dùng
$posts = $country->posts;
foreach ($posts as $post) {
    echo $post->title;
}
```

Tùy chỉnh khóa ngoại và khóa chính trong hasManyThrough

Theo mặc định, Laravel sẽ giả định rằng:

- Khóa ngoại trên model trung gian (users) là country_id
- Khóa ngoại trên model cuối (posts) là user_id
- Khóa chính của model Country là id.

Nếu tên các cột trong bảng của bạn khác với mặc định, bạn có thể chỉ định rõ ràng các cột này khi định nghĩa quan hệ hasManyThrough().

```php
public function posts()
{
    return $this->hasManyThrough(
        Post::class,       // Model cuối cùng
        User::class,       // Model trung gian
        'country_id',      // Khóa ngoại trên bảng trung gian (users)
        'user_id',         // Khóa ngoại trên bảng cuối cùng (posts)
        'id',              // Khóa chính trên bảng gốc (countries)
        'id'               // Khóa chính trên bảng trung gian (users)
    );
}
```

loadCount

Trong Laravel Eloquent, phương thức loadCount() cho phép bạn tính toán số lượng bản ghi của các mối quan hệ (relationship) sau khi model chính đã được truy xuất từ cơ sở dữ liệu. Đây là lý do vì sao nó còn được gọi là Deffered hay Lazy, vì ban đầu khi truy vấn model chính, Laravel sẽ không thực hiện lệnh count các relation. Laravel sẽ chỉ thực hiện tại thời điểm bạn gọi loadCount() mà thôi. Điều này sẽ có lợi, vì không phải lúc nào bạn cũng cần count ngay số lượng các bản ghi tại thời điểm truy vấn như withCount(). Ngoài ra nó cũng giúp bạn linh hoạt hơn khi xử lý các tình huống nghiệp vụ cần đếm số lượng.

Khi bạn gọi phương thức loadCount():

- Laravel sẽ thực hiện một truy vấn riêng biệt để đếm số lượng bản ghi trong relation mà bạn yêu cầu.
- Kết quả count sẽ được lưu vào một thuộc tính đặc biệt theo định dạng relationship_count. Ví dụ, nếu bạn đếm số lượng bình luận (comments) của một bài viết (post), kết quả sẽ được lưu vào thuộc tính comments_count.

Cú pháp của loadCount cũng rất đơn giản:

```
$model->loadCount('relationship');
```

Giả sử bạn có model Post và mối quan hệ comments (một bài viết có nhiều bình luận). Bạn có thể sử dụng loadCount() để đếm số lượng bình luận của một bài viết mà không cần load toàn bộ các bình luận.

```
$post = Post::find(1);  // Lấy bài viết từ cơ sở dữ liệu

// Tính số lượng bình luận của bài viết
$post->loadCount('comments');

echo $post->comments_count;  // Hiển thị số lượng bình luận của bài viết
```

Sử dụng loadCount() với điều kiện

Bạn cũng có thể thêm điều kiện khi đếm số lượng bản ghi trong relation. Ví dụ, bạn chỉ muốn đếm những bình luận đã được phê duyệt:

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

// Tính số lượng bình luận đã được phê duyệt
$post->loadCount(['comments' => function ($query) {
    $query->where('approved', true);
}]);

echo $post->comments_count;  // Hiển thị số lượng bình luận đã được phê duyệt
```

Aggregate functions

Laravel Eloquent từ phiên bản 9.x đã bổ sung các hàm tổng hợp (aggregate functions) withMin, withMax, withAvg, withSum, và withExists, giúp bạn dễ dàng tính toán các giá trị tổng hợp khi truy vấn các relation của model mà không cần viết các truy vấn SQL thủ công hoặc thực hiện tổng hợp dựa trên kết quả get() ra nữa.
Mục đích của các hàm này:

- withMin, withMax, withAvg, và withSum được sử dụng để thực hiện các phép tính tổng hợp (aggregate) trên một relation, ví dụ như tìm giá trị nhỏ nhất, giá trị lớn nhất, trung bình, hay tổng của một cột trong relation đó.
- withExists được sử dụng để kiểm tra sự tồn tại của các bản ghi trong một mối quan hệ và trả về true hoặc false.

Giả sử bạn có ba model User, Post, và Comment:

- Một User có nhiều Post.
- Một Post có nhiều Comment.

Bạn có thể sử dụng các hàm này để tính toán các giá trị tổng hợp dựa trên mối quan hệ giữa posts và comments.

- withMin(): Tìm giá trị nhỏ nhất

Ví dụ bạn muốn tìm bài viết với số lượng bình luận nhỏ nhất cho mỗi người dùng:

```
$users = User::withMin('posts', 'comments_count')->get();

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

- withMax(): Tìm giá trị lớn nhất

Bạn có thể tìm bài viết có nhiều bình luận nhất cho mỗi người dùng:

```
$users = User::withMax('posts', 'comments_count')->get();
foreach ($users as $user) {
    echo $user->posts_max_comments_count;
}
```

- withAvg(): Tính giá trị trung bình

Để tính số bình luận trung bình cho mỗi bài viết của một người dùng, bạn có thể sử dụng withAvg():

```
$users = User::withAvg('posts', 'comments_count')->get();
foreach ($users as $user) {
    echo $user->posts_avg_comments_count;
}
```

- withSum(): Tính tổng

Để tính tổng số bình luận cho tất cả bài viết của một người dùng, bạn có thể sử dụng withSum():

```
$users = User::withSum('posts', 'comments_count')->get();
foreach ($users as $user) {
    echo $user->posts_sum_comments_count;
}
```

- withExists(): Kiểm tra sự tồn tại

Bạn có thể kiểm tra xem người dùng có bài viết nào không:

```
$users = User::withExists('posts')->get();
```

```
foreach ($users as $user) {
    echo $user->posts_exists ? 'Has Posts' : 'No Posts';
}
```

Tương tự như loadCount(), chúng ta cũng có các Deffered hay Lazy Function cho các phương thức này như loadMin(), loadMax(), loadAvg(), ….

Eloquent Collection

Eloquent collections là các instance của lớp Illuminate\Database\Eloquent\Collection. Chúng là phần mở rộng của Laravel Collection, cho phép bạn sử dụng tất cả các phương thức của Collection thông thường. Ngoài ra, Eloquent collections cung cấp thêm nhiều phương thức đặc biệt để làm việc với các model Eloquent. Dưới đây là một số helper của Eloquent Collection mình thấy hay và mọi người có thể tận dụng trong dự án (Lưu ý, chúng không phải toàn bộ các helper của Eloquent Collection, mà là các helper mình chọn lọc. Bạn có thể tham khảo tất cả chúng tại đây):

load($relations)

Eager load relation mà bạn muốn cho các model trong collection.

```
$users->load(['comments', 'posts']);

$users->load('comments.author');

$users->load(['comments', 'posts' => fn ($query) => $query->where('active', 1)]);
```

loadMissing($relations)

Eager load relation mà bạn muốn cho các model trong collection trong trường hợp chúng chưa được load trước đó.

```
$users->loadMissing(['comments', 'posts']);

$users->loadMissing('comments.author');

$users->loadMissing(['comments', 'posts' => fn ($query) => $query->where('active', 1)]);
```

toQuery()

Chuyển đổi collection thành một truy vấn Eloquent mới. Về bản chất, Laravel sẽ chạy câu truy vấn whereIn các bản ghi trong Collection theo primaryKey rồi trả ra một instance Eloquent Query Builder để bạn có thể thao tác tiếp

```
use App\Models\User;

$users = User::where('status', 'VIP')->get();

$users->toQuery()->update([
   'status' => 'Administrator',
]);
```

setHidden($attributes)

setHidden($attributes) trong Eloquent Collections cho phép tạm thời ghi đè tất cả các thuộc tính bị ẩn (hidden attributes) trên từng model trong collection. Điều này có thể hữu ích khi bạn muốn tùy chỉnh thuộc tính nào sẽ được ẩn khi trả về một array hoặc JSON mà không cần tác động tới Model. Laravel có hàm ngược lại tương tự là makeHidden().

```
$users = $users->setHidden(['email', 'password', 'remember_token']);
```

setVisible($attributes)

Phương thức setVisible($attributes) tạm thời ghi đè tất cả các thuộc tính có thể hiển thị (visible attributes) trên từng model trong collection. Laravel có hàm ngược lại tương tự là makeVisible().

```
$users = $users->setVisible(['id', 'name']);
```

fresh($with = [])

Phương thức fresh() trong Eloquent Collection sẽ lấy một instance mới từ database cho từng model trong collection, giúp đảm bảo dữ liệu được cập nhật và mới nhất. Ngoài ra, bạn có thể chỉ định các relation sẽ được eager load cùng với dữ liệu của model.

```
$users = $users->fresh();

$users = $users->fresh('comments');
```

Lời kết

Bạn đã đọc hết cuốn “Eloquent: Có thể bạn chưa biết - Những tips làm việc cùng Eloquent” của mình.
Từ đáy lòng, mình cám ơn bạn rất nhiều vì đã đọc đến cuối quyển Ebook này. Có thể cách viết mình còn lộn xộn, mình mong bạn hiểu những tâm huyết của mình, và những mong mỏi được chia sẻ kiến thức đến cộng đồng. Làm việc với Eloquent của Laravel là điều rất thú vị, nhưng cũng tiềm ẩn rất nhiều thứ cần cẩn trọng. Nếu tận dụng tốt, nó sẽ rất mạnh mẽ, ngược lại chúng ta cũng dễ mắc những lỗi cơ bản. Mình hy vọng ebook này sẽ giúp đỡ bạn phần nào trên con đường học lập trình nhé .
Cám ơn bạn rất nhiều !
Nếu bạn thích những gì trong cuốn sách này và muốn trao đổi thêm, đừng ngần ngại liên lạc với mình qua:
Email: huynt57@gmail.com
Facebook: Tại đây

Cuốn Ebook này sẽ không tồn tại nếu không có vợ mình Dương Thị Thu Huyền và con trai mình, cố vấn tí hon Nguyễn Dương Hoàng Khôi. Cám ơn gia đình đã luôn bên cạnh và ủng hộ mình.

Happy Coding !