**BỘ GIÁO DỤC VÀ ĐÀO TẠO**

**TRƯỜNG ĐẠI HỌC MỞ THÀNH PHỐ HỒ CHÍ MINH**



**TÌM HIỂU LARAVEL FRAMEWORK**

**XÂY DỰNG WEBSITE BÁN HÀNG**

**(Phần Mô Tả Kĩ Thuật / Công Nghệ Sử Dụng)**

**MÔN**

**LẬP TRÌNH CƠ SỞ DỮ LIỆU**

**IT81 - Nhóm 7**

**Trịnh Thị Kim Vui 1851010157**

**Lý Quốc Dũng 1851050025**

**Trần Minh Trí 1851050157**

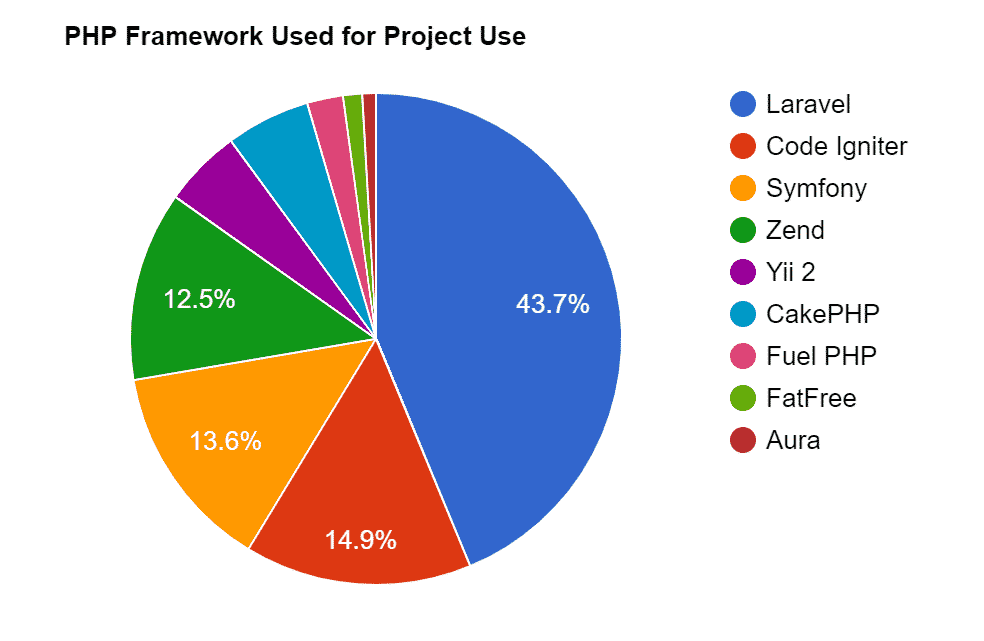
**Giảng viên hướng dẫn: ThS. Lê Ngọc Hiếu**

**TP. HỒ CHÍ MINH, 2021**

# Tổng quan về Laravel Framework

Thuật ngữ Framework được dùng để chỉ các đoạn mã lệnh được viết sẵn, các thư viện chứa tập hợp các file, các hàm cơ bản được phát triển và cung cấp bởi các công ty lập trình. Framework cung cấp cho các lập trình viên những chức năng cơ bản nhất của một ứng dụng web. Khi lập trình với các framework, ta không cần phải tự viết các mã lệnh thông dụng đã được dùng nhiều lần. Từ đó giúp việc lập trình trở nên dễ dàng, đồng thời tuân thủ theo các quy tắc chung của một framework sẽ giúp ta có hướng đi rõ ràng và tốt hơn.

PHP là một trong những ngôn ngữ lập trình web phổ biến trên thế giới với một cộng đồng lớn các nhà phát triển. Với sức mạnh cộng đồng này, các PHP framework đang được phát triển mạnh mẽ, vượt trội và được sử dụng vô cùng rộng rãi.



Hình 1.1: Độ phổ biến của các PHP framework

Taylor Otwell cho ra mắt phiên bản Laravel đầu tiên vào tháng 6 năm 2011. Laravel được thừa hưởng những ưu điểm, tính năng mạnh mẽ từ các framework khác như Console Component của Symfony. Đồng thời, Laravel được tạo ra nhằm cung cấp giải pháp thay thế nâng cao thay cho framework CodeIgniter; hỗ trợ thêm các tính năng bảo mật như xác thực (authentication) và ủy quyền (authorization). Laravel hỗ trợ rất tốt trong quá trình xây dựng các ứng dụng web theo mô hình MVC.

Sau hơn 20 năm phát triển, Laravel đã phát hành đến phiên bản thứ 8 với những tính năng mạnh mẽ, vượt qua những framework khác và vươn lên trở thành framework phổ biến và được cộng đồng các nhà phát triển PHP sử dụng nhiều nhất.

Ngoài ra, phiên bản Laravel thứ 9 sẽ được cho ra mắt vào nửa đầu năm 2022 với những tính năng ưu việc khác. Đồng thời, đây là phiên bản đầu tiên sau Laravel 6 được hỗ trợ dài hạn với LTS (Long Term Support).

## Lí do nên chọn Laravel

Laravel không chỉ thừa hưởng những ưu điểm mà còn khắc phục được những hạn chế của các PHP framework trước đó, giúp Laravel trở thành lựa chọn tốt nhất cho các nhà phát triển ứng dụng web.

**Khả năng phát triển**

Laravel luôn đồng hành với các nhà phát triển trong quá trình phát triển ứng dụng cũng như kĩ năng. Với các nhà phát triển mới, Laravel cung cấp các tài liệu, hướng dẫn cơ bản vô cùng chi tiết để bắt đầu. Đồng thời, Laravel cũng cung cấp các công cụ kĩ thuật nâng cao như dependency injection, kiểm thử với unit testing, queues, sự kiện real-time để phát triển ứng dụng web chuyên nghiệp.

**Khả năng mở rộng**

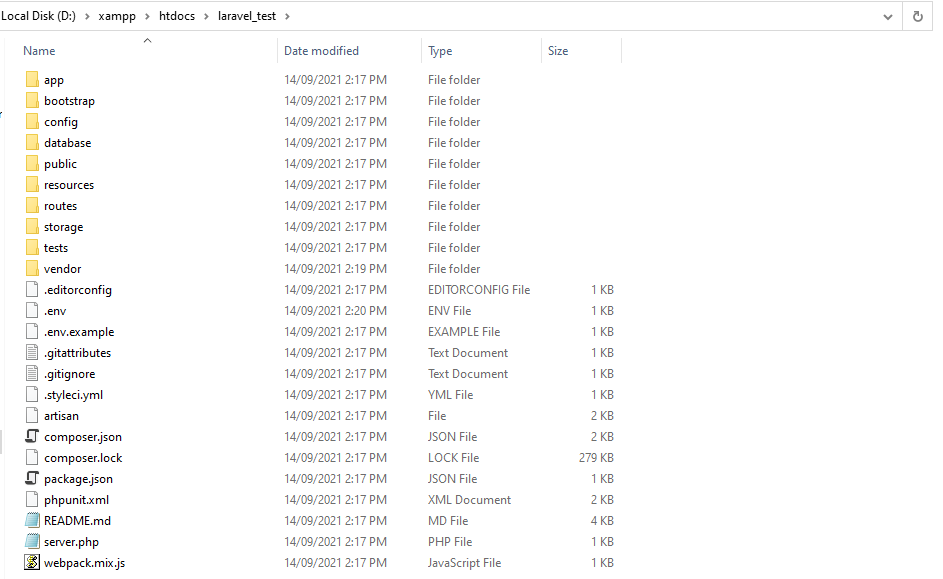
Laravel có khả năng mở rộng đáng kinh ngạc. Nhờ tính chất scaling-friendly của PHP và khả năng built-in của mình, Laravel hỗ trợ nhanh chóng và mạnh mẽ các hệ thống bộ nhớ cache phân tán như **Redis**. Trên thực tế, các ứng dụng Laravel dễ dàng được mở rộng để xử lý hàng trăm triệu request mỗi tháng. Ngoài ra, các nền tảng như **Laravel Vapor** cho phép ta chạy ứng dụng Laravel của mình ở quy mô gần như vô hạn trên công nghệ không máy chủ mới nhất của **AWS**.

**Framework cộng đồng**

Với sự đóng góp của hàng nghìn nhà phát triển tài năng từ khắp nơi trên thế giới, ta có thể dễ dàng tìm được sư hướng dẫn từ bất kì đâu.

## Cấu trúc thư mục

Để hiểu thêm về Laravel, ta hãy xem qua cấu trúc tổ chức thư mục trong Laravel phiên bản 8:



Hình 1.2: Cấu trúc thư mục trong Laravel

Trong đó:

* *App*: Gồm các thư mục chứa Controller, Models, Request của ứng dụng.
* *Boostrap:* Chứa một số file của Boostrap Framework và một số file cấu hình nạp tự động.
* *Config*: Chứa các file cấu hình hệ thống như ứng dụng, cơ sở dữ liệu, view...
* *Database*: Gồm các thư mục chứa Factories, Migrations và Seeders để quản lí các phiên bản của cơ sở dữ liệu.
* *Public*: Thường được dùng để chứa các loại file như hình ảnh, css, js của ứng dụng. Trang index.php cũng nằm trong thư mục này.
* *Resources:* Gồm các thư mục chứa các view, raw asset và các file ngôn ngữ (lang).
* *Routes:* Chứa các file lưu định tuyến giữa URL với các Controller, API...
* *Storage:* Lưu trữ file tạm thời cho các dịch vụ như Session, Cache… Thư mục này có thể được ghi lại bởi web server và được duy trì bởi Laravel mà không cần ta can thiệp.
* *Tests:* Chứa các test kiểm thử tự động.
* *Vendor:* Chứa các code của bên thứ ba, các plugin cài thêm và các gói composer.

# Mô hình MVC

## Tổng quan MVC

Model – View – Controller (MVC) là một mô hình kiến trúc quan trọng trong khoa học máy tính và rất thích hợp cho các ứng dụng web. Mô hình MVC phân tách dữ liệu và logic nghiệp vụ khỏi giao diện hiển thị. Một số MVC framework phổ biến hiện nay là Laravel, CodeIgniter (PHP), Spring (Java), ASP.NET MVC.

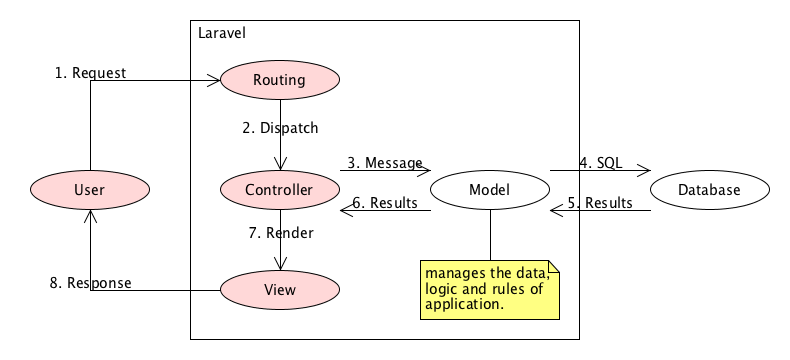
Chức năng của các thành phần MVC:

* Model: tập hợp các lớp mô tả dữ liệu và các quy tắc, thao tác dữ liệu.
* View: xác định cách hiển thị và cung cấp giao diện tương tác với người dùng.
* Controller: tập hợp các lớp xử lí giao tiếp từ người dùng, luồng ứng dụng tổng thể và logic dành riêng cho ứng dụng. Controller hoạt động giữa Model và View.

## Hoạt động của MVC trong Laravel

Laravel là framework hỗ trợ mạnh mẽ mô hình MVC. Ngoài các thành phần chính là Model, View, Controller, Laravel còn có thêm Route để định tuyến đến đúng Controller theo địa chỉ URL mà người dùng nhập.

Khi người dùng gửi yêu cầu, Route sẽ định hướng yêu cầu đến đúng Controller. Controller sẽ gọi Model để xử lí các yêu cầu. Model thao tác dữ liệu với cơ sở dữ liệu và nhận kết quả trả về để gửi cho Controller. Controller xuất kết quả nhận được cho View để View gửi kết quả phản hồi cho người dùng.



Hình 2.1: Sơ đồ hoạt động mô hình MVC trong Laravel

# Artisan Console

Được phát triển dựa trên component của Symfony Framework, Artisan Console là một giao diện dòng lệnh (command-line interface) được tích hợp sẵn trong thư mục gốc ứng dụng Laravel dưới dạng tập lệnh tên artisan.

Là một “nghệ nhân”, Artisan Console xuất sắc trong việc xử lí các vấn đề thủ công một cách tự động hóa. Không chỉ cung cấp các lệnh hữu ích trong quá trình xây dựng ứng dụng như tự động sinh key mã hóa trong xác thực OAuth2, Artisan còn có thể tránh được việc thường xuyên lặp đi lặp lại hầu hết các tác vụ trong quá trình phát triển như thao tác với cơ sở dữ liệu (migrate, seed...), tạo các template chức năng giống nhau cho Model, Controller... từ đó giúp các nhà phát triển rút ngắn thời gian xây dựng đồng thời tăng hiệu suất làm việc.

Để xem danh sách lệnh artisan, ta mở command trên thư mục gốc của ứng dụng và sử dụng câu lệnh sau:

php artisan list

**Một số lệnh artisan đơn giản**

# Xem các trợ giúp

php artisan help migrate

# Tạo một Controller có tên là UserController

php artisan make:controller UserController

# Tạo một Model có tên là User

php artisan make:model User

# Tạo migration

php artisan make:migration create\_user\_table

# Tạo factory

php artisan make:factory UserFactory

# Tạo seeder

php artisan make:seeder UserSeeder

# Tạo model kèm migration, factory, seeder, và controller

php artisan make:model User -mfsc

# Routing

Routing trong Larvel có chức năng định tuyến các action cho các sự kiện (event) tương ứng. Laravel hỗ trợ mạnh mẽ định tuyến và cung cấp 3 loại route, gồm: route dành cho giao diện web, route dành cho CLI (commnad) và route dành cho Broadcast.

Theo mặc định, các route trong Lavel được khai báo trong các file định tuyến nằm trong thư mục routes. Các file này sẽ được tải tự động bởi ứng dụng qua App\Providers\RouteServiceProvider. Nếu muốn khai báo route trong file khác, cần định nghĩa nó trong RouteServiceProvider.

Ta có thể khai báo route trong routes/web.php hoặc routes/api.php. Trong đó, routes/web.php định nghĩa các router được dùng trong giao diện web của ứng dụng. Các route này được gán cho nhóm phần mềm trung gian web, cung cấp các tính năng như trạng thái session và CSRF. Nếu khai báo route trong routes/api.php, các đường dẫn sẽ được thêm tiền tố api ở phía trước.

## Các phương thức HTTP request

|  |  |
| --- | --- |
| **Tên phương thức HTTP request** | **Phương thức nhận request** |
| Route::get($uri, $callback); | GET |
| Route::post($uri, $callback); | POST |
| Route::put($uri, $callback); | PUT |
| Route::patch($uri, $callback); | PATCH |
| Route::delete($uri, $callback); | DELETE |
| Route::options($uri, $callback); | OPTIONS |

Bảng 2.2: Các phương thức HTTP request

Route sẽ nhận vào đường dẫn $uri cần xử lí và nhận $callback có thể là callback, callable hoặc một array chứa thông tin về controller và phương thức cần phải thực hiện.

Ta cũng có thể đăng kí một route hỗ trợ nhiều phương thức HTTP bằng cách sử dụng phương thức match với $methods là một mảng chứa các phương thức mà route cần xử lí.

Route::match($methods, $uri, $callback);

Hoặc nhận tất cả các phản hồi từ các phương thức HTTP bằng phương thức any:

Route::any($uri, $callback);

## Route redirect và Route view

Để chuyển hướng route đến URI khác mà không cần chỉnh sửa route hay controller, ta sử dụng phương thức redirect. redirect nhận tham số thứ nhất chứa đường dẫn URI muốn xử lí và tham số thứ hai chứa đường dẫn muốn chuyển hướng đến:

Route::redirect($uri, $redirectTo);

Route::redirect(‘/parent’, ‘/child’);

Mặc định, redirect sẽ trả về HTTP status là 302. Để chỉnh HTTP status, ta thêm vào tham số thứ ba chứa HTTP status cần trả về.

Route::redirect($uri, $redirectTo, $status);

Trong trường hợp chỉ cần trả về một view mà không cần xác định toàn bộ route hay controller, ta sử dụng phương thức view. Tương tự redirect, view nhận tham số thứ nhất chứa đường dẫn URI muốn xử lí và tham số thứ hai chứa là tên view cần trả về. Trường hợp cần truyền dữ liệu vào view, ta thêm tham số thứ ba là mảng chứa dữ liệu cần thêm:

Route::view($uri, $viewName);

Route::view($uri, $viewName, $data);

Ta cũng có thể truyền thêm hai tham số HTTP là status và header vào route view.

*Chú ý: Route view chỉ hoạt động với phương thức GET hoặc HEAD.*

## Truyền tham số trong Route

Để route có thể nhận vào tham số, cần thêm cú pháp {name} vào $uri với name là định danh của tham số đó, ví dụ:

Route::get('user/{id}', function ($id) {

return "User id: $id";

});

Trường hợp cần gán nhiều tham số vào URI, ta thêm từng tham số như sau:

Route::get('/posts/{post}/comments/{comment}',

function ($postId, $commentId) {

//

});

Nếu tham số là không bắt buộc, ta thêm ‘?’ vào sau tên tham số đồng thời khởi gán giá trị mặc định cho callback.

Route::get('user/{id?}', function ($id = null) { 🡸

if (!$id) {

return "Input your id!";

}

return "User id: $id";

});

## Định dạng tham số

Để định dạng dữ liệu khi truyền vào route, ta có thể sử dụng phương thức where. Với where, ta ó thể định dạng một hoặc nhiều tham số trong môt mảng.

Route::get('/user/{id}', function ($id) {

//

})->where('id', '[0-9]+');

Route::get('/user/{id}/{name}', function ($id, $name) {

//

})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);

Ngoài where, Laravel cũng hỗ trợ thêm một số phương thức định dạng dữ liệu khác như whereAlpha(), whereNumber(), whereAlphaNumeric(), whereUuid().

Để việc định dạng dữ liệu truyền vào route hoạt động tốt nhất, ta có thể sử dụng phương thức Route::pattern để chứa các mẫu định dạng. Laravel khuyến khích nên đặt pattern bên trong phương thức boot của RouteServiceProvider. Ví dụ:

public function boot() {

Route::pattern('id', '[0-9]+');

}

## Đưa phương thức trong controller vào route

Ta có thể gọi phương thức trong controller bằng cú pháp:

Route::routeMethod($uri, [$controllerName, $method]);

Trong đó:

* routeMethod: tên phương thức HTTP request.
* $uri: đường dẫn muốn xử lí.
* $controllerName: tên controller.
* $method: tên phương thức trong controller.

Ví dụ: Gọi phương thức greet trong GreetingController cho URI welcome:

Route::get(‘/welcome’,

[GreetingController::class, 'greet']);

## Nhóm các route

Để quản lí route có nhiều thuộc tính giống nhau, Laravel hỗ trợ phương thức group để gom nhóm các route.

Trường hợp các router có cùng middleware, ta có thể sử dụng phương thức middleware trước khi gom nhóm:

Route::middleware(['first', 'second'])->group(function () {

Route::get('/', function () {

// Dùng middleware first & second

});

Route::get('/user/profile', function () {

// Dùng middleware first & second

});

});

Phương thức group cũng hỗ trợ gom nhóm các route có cùng domain. Ta cần gọi phương thức domain để chọn các domain trước khi gom nhóm:

Route::domain('{account}.example.com')->group(function () {

Route::get('user/{id}', function ($account, $id) {

//

});

});

Phương thức prefix giúp xác định tiền tố các URI có tiền tố giống nhau, ta cũng có thể gom nhóm các prefix này như sau:

Route::prefix('user')->group(function () {

Route::get('name', function () {

//

});

Route::get('password', function () {

//

});

});

## Đặt tên cho route

Để việc chuyển hướng route diễn ra thuận tiện, ta có thể đặt tên cho route bằng cách sử dụng phương thức name($name) hoặc as($name), ví dụ:

Route::get(‘/User’, [UserController::class, 'login'])

->name(‘login’);

Vì mỗi route là duy nhất nên tên của các route cũng không được trùng với nhau. Ngoài ra, ta cũng có thể đặt tên cho các prefix và route bên trong prefix:

Route::prefix('user')->as('user.')->group(function () {

Route::get('name', function () {

//

})->name('name');

Route::get('password', function () {

//

})->name('password');

});

Generate URL qua phương thức route sẽ có dạng:

route(‘login’);

// http://example.com/login

route('user.name');

// http://example.com/user/name

# Controller

Laravel hỗ trợ mạnh mẽ mô hình MVC, Controller thuộc C trong MVC. Đúng như tên gọi, controller chính là một bộ điều khiển, nó chứa tất cả các hợp các lớp xử lí giao tiếp từ người dùng, các luồng ứng dụng tổng thể và logic dành riêng cho ứng dụng. Controller hoạt động giao tiếp giữa Model và View.

Để một controller hoạt động được, cần tuân thủ các quy tắc sau:

* Controller phải được đặt trong đường dẫn App\Http\controllers.
* Tên controller phải giống với tên class trong file controller đó.
* Class controller vừa tạo phải kế thừa (extends) class Controller.

## Tạo Controller

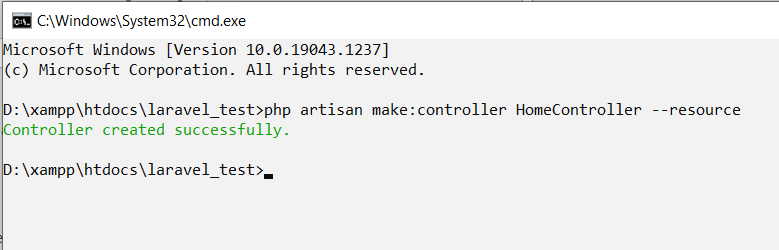
Ta có thể truy cập đường dẫn App\Http\controllers để tạo controller theo cách thủ công. Hoặc vào thư mục gốc của dự án và mở Command Line và sử dụng một trong hai câu lệnh sau để tạo tự động một controller với ControllerName là tên controller cần tạo:

php artisan make:controller ControllerName

// hoặc

php artisan make:controller ControllerName --resource

Ví dụ:



Hình 5.1: Tạo controller tự động

Với câu lệnh đầu, Laravel sẽ tạo controller trống như sau:

<?php

namespace App\Http\Controllers;

use App\User;

use App\Http\Controllers\Controller;

class HomeController extends Controller

{

//

}

Với câu lệnh còn lại, Laravel sẽ tạo controller có sẵn phương thức index và các phương thức theo chu trình CRUD (create, read, update, delete) như sau:

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class HomeController extends Controller

{

public function index()

{

//

}

public function create()

{

// Hiện trang thêm dữ liệu

}

public function store(Request $request)

{

// Thêm dữ liệu

}

public function show($id)

{

// Hiển thị dữ liệu theo $id

}

public function edit($id)

{

// Hiện trang sửa dữ liệu theo $id

}

public function update(Request $request, $id)

{

// Cập nhật dữ liệu

}

public function destroy($id)

{

// Xóa dữ liệu

}

}

## Sử dụng action trong Controller

Để sử dụng một action trong controller, ta cần tạo một hàm trong controller đó và gọi hàm đó trong route. Ví dụ,tạo hàm index trong HomeController:

<?php

namespace App\Http\Controllers;

class HomeController extends Controller

{

public function index()

{

echo "Index content!";

}

}

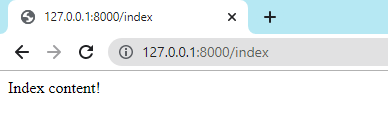
Và gọi trong route:

use App\Http\Controllers\HomeController;

Route::get('index', [HomeController::class,'index']);

//Route::get('index', 'HomeController@index');

Kết quả:



Hình 5.2: Sử dụng action trong Controller

## Truyền tham số trong Controller

Truyền tham số cho controller trong Laravel cũng tương tự như truyền tham số cho controller trong PHP thuần.

Ví dụ, thêm biến $name vào hàm hello trong HomeController như sau:

<?php

namespace App\Http\Controllers;

class HomeController extends Controller

{

public function hello($name){ 🡸

echo "Hello, $name!!!";

}

}

Tạo route cho action gọi hello() với {name} là tham số cần truyền:

Route::get('hello/{name}', 🡸

[HomeController::class,'hello']);

## Middleware Controller

Ta có thể gán Middleware cho Controller trong route như sau:

Route::get('profile', 'UserController@show')

->middleware('auth'); 🡸

Tuy nhiên, sẽ tiện lợi hơn khi ta định nghĩa middleware trong phương thức khởi tạo (contructor) của controller. Sử dụng phương thức middleware theo cách này giúp ta dễ dàng gán middleware cho controller. Ngoài ra, ta cũng có thể hạn chế middleware cho một vài phương thức cụ thể trong controller:

class HomeController extends Controller

{

public function \_\_construct()

{

$this->middleware('auth');

$this->middleware('log')->only('index');

$this->middleware('subscribed')->except('store');

}

}

# View

Theo mặc định, các view sẽ được đặt trong thư mục resources/views. Ta có thể vào config/view.php và thay đổi giá trị paths để thay đổi thư mục chứa view. Laravel hỗ trợ view có phần mở rộng là .html, .css, .php, và .blade.php. Trường hợp trong thư mục có các view trùng tên, Laravel sẽ ưu tiên các file theo thứ tự: .blade.php, .php, .css và .html.

## Render view

Để render view trong Laravel, sử dụng cú pháp:

view($view, $data);

// hoặc

use \Illuminate\Support\Facades\View;

View::make($view, $data);

Trong đó:

* $view: đường dẫn đến view (tính từ thư mục views).
* $data: mảng dữ liệu truyền vào trong view, tham số này có thể trống.

Ví dụ, tạo trang [home.blade.php](http://home.blade.php) có nội dung như sau:

<!doctype html>

<html>

<head>

<title>Laravel Test</title>

</head>

<body>

<h1>Hello world!</h1>

</body>

</html>

Sau đó ta khai báo route home để hiển thị home.blade.php:

/\* routes/web.php \*/

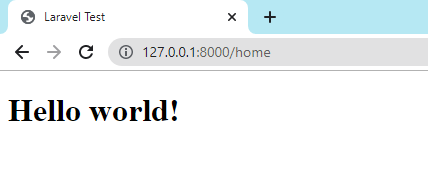
use Illuminate\Support\Facades\Route;

Route::get('/home', function () {

return view('home');

})->name('home');

Tạo route child và truy cập route đó, kết quả:



Hình 6.1: Ví dụ render view trong Laravel

## View first

Ta có thể sử dụng phương thức first trong đối tượng View để render view tồn tại đầu tiên trong danh sách view với cú pháp:

use Illuminate\Support\Facades\View;

View::first($views, $data);

Trong đó:

* $views: một array các đường dẫn đến view (tính từ thư mục views).
* $data: array data muốn truyền vào trong view. Tham số này có thể bỏ trống.

**Kiểm tra view tồn tại**

Để kiểm tra một view có tồn tại hay không, sử dụng phương thức exists với $view là đường dẫn đến view muốn kiểm tra:

use Illuminate\Support\Facades\View;

View::exists($view):

Ví dụ, kiểm tra xem thư mục views có view tên home hay không:

use Illuminate\Support\Facades\View;

if (View::exists('home')) {

// Code xử lí khi tồn tại view home

}

## Truyền dữ liệu cho tất cả view

Trong Laravel, ta có thể sử dụng phương thức share trong đối tượng View để truyền dữ liệu cho tất cả các view. Để ứng dụng hoạt động tốt và chính xác nhất, nên thêm View::share vào Providers/AppServiceProvider.php

Ví dụ, share dữ liệu có name là siteOwner cho tất cả các view:

<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider

{

public function boot()

{

View::share('siteOwner', 'Vui Trinh');

}

}

Ta có thể sử dụng biến $siteOwner trong các view. Ngoài ra, ta cũng có thể sử dụng View::composer để truyền một đoạn code dùng chung, tránh lặp lại code.

# Blade Template

Blade là một công cụ tạo mẫu (templating engine) giúp xử lí cú pháp trong view một cách ngắn gọn và logic. Không như các template khác, Blade cho phép ta sử dụng cả PHP thuần bên trong. Tất cả các Blade Template sẽ được biên dịch về mã lệnh PHP thuần và lưu vào bộ nhớ đệm. Về cơ bản, sử dụng Blade Template không khiến ứng dụng hoạt động chậm đi. Các file Blade View có phần mở rộng là .blade.php.

## Hiển thị dữ liệu

Để hiển thị dữ liệu trong Blade, ta đặt mã lệnh vào cặp {{}}. Ví dụ:

* routes/web.php

Route::get('/home', function() {

return view('home',

['heading1' => "<h1>Laravel Test - Title</h1>"]);

});

* Trang blade.view.php:

<!doctype html>

<html>

<head>

<title>Laravel Test</title>

</head>

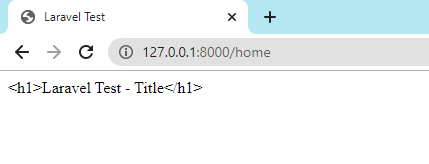
<body>

{{$heading1}}

</body>

</html>

Kết quả:



Hình 7.1: Ví dụ hiển thị dữ liệu trong Blade

Ta thấy không thể in ra mã HTML khi sử dụng cặp dấu {{}} này. Để in mã HTML ta sử dụng {!!!!}, thay đoạn body trong trang blade.view.php như sau:

<body>

{!! $heading1 !!}

</body>

Kết quả:



Hình 7.2: Ví dụ hiển thị dữ liệu trong Blade

## Kế thừa bố cục

Trong quá trinh xây dựng ứng dụng web, thường có các layout được sử dụng trên nhiều trang khác nhau. Để tranh việc lặp đi lặp lại code, ta có thể sử dụng kế thừa thành phần và kế thừa mẫu. Đây là hai trong số những lợi ích chính của việc sử dụng Blade.

Ví dụ: Tạo trang home.blade.php:

<!doctype html>

<html>

<head>

<title>Laravel Test - @yield('title')</title>

</head>

<body>

@section('headding1')

<h1>Hello World!</h1>

@show

</body>

</html>

Trong đó:

* @yield: hiển thị nội dung của một phần nhất định.
* @section: đánh dấu bắt đầu một phần nội dung hiển thị của trang.
* @show: đánh dấu kết thúc một phần nội dung hiển thị của trang.

Tạo thêm trang child.blade.php và kế thừa nội dung:

@extends('home')

@section('title', 'Page Title')

@section('heading1')

@parent

<p>This is a paragraph</p>

@endsection

Trong đó:

* @endsection: đánh dấu kết thúc một phần nội dung hiển thị của trang.
* @parent: hiển thị nội dung cha.

Kết quả:



Hình 7.3: Trang child.blade.php

## Cấu trúc lựa chọn switch…case

Ta sử dụng @switch, @case, @break, @default và @endswitch tương tự switch, case, break, default và endswitch trong PHP.

@switch($i)

@case(1)

First case...

@break

@case(2)

Second case...

@break

@default

Default case...

@endswitch

## Cấu trúc điều kiện if

Ta sử dụng @if, @else, @elseif và @endif tương tự if, else, elseif và endif trong PHP.

@if (count($records) === 1)

I have one record!

@elseif (count($records) > 1)

I have multiple records!

@else

I don't have any records!

@endif

Blade Template cũng hỗ trợ các directive rút gọn khác như @unless, @isset, @empty, @guest, @env…

## Các cấu trúc vòng lặp

Với cấu trúc vòng lặp, Laravel không hỗ trợ do-while, bên dưới các driective được hỗ trợ:

@for ($i = 0; $i < 10; $i++)

The current value is {{ $i }}

@endfor

@foreach ($users as $user)

<p>This is user {{ $user->id }}</p>

@endforeach

@forelse ($users as $user)

<li>{{ $user->name }}</li>

@empty

<p>No users</p>

@endforelse

@while (true)

<p>I'm looping forever.</p>

@endwhile

Ta cũng có thể lồng cấu trúc điều kiện vào cấu trúc vòng lặp và ngược lại. Ví dụ:

@foreach ($users as $user)

@if ($user->type == 1)

@continue

@endif

@endforeach

# Database

Laravel hỗ trợ mạnh mẽ cơ sở dữ liệu qua các lớp **Query Builder** và **Eloment ORM**. Đồng thời Laravel cũng hỗ trợ nhiều cơ sở dữ liệu và nhiều kết nối trên cùng một ứng dụng. Mặc định, Laravel hỗ trợ các loại cơ dữ liệu sau:

* MySQL 5.6+
* PostgreSQL 9.4+
* SQLite 3.8.8+
* SQL Server 2017+

Ngoài ra, ta cũng có thể sử dụng các cơ sở dữ liệu khác từ các package cung cấp bởi các bên thứ ba.

## Cấu hình

Tất cả các thông tin cấu hình database trong Laravel đều được đặt trong file config/database.php. File này cũng chứa thông tin kết nối đến nhiều database, ta cấu hình connection mặc định tại đây. Thông thường, thông tin cấu hình đã được Laravel mặc định có thể sẵn sàng kết nối với **Laravel Sail**.

**SQLite**

SQLite database được chứa trong một file duy nhất trên hệ thống. Ta có thể tạo ra một SQLite database mới bằng lệnh touch trong command:

touch database/database.sqlite.

Khi đã tạo cơ sở dữ liệu, ta có thể dễ dàng cấu hình các biến môi trường cách đặt đường dẫn tuyệt đối đến database cho DB\_DATABASE trong file .env.

DB\_CONNECTION=sqlite

DB\_DATABASE=/absolute/path/to/database.sqlite

Để bật các ràng buộc khóa ngoại (foreign key) cho các kết nối SQLite, nên đặt true cho DB\_FOREIGN\_KEYS:

DB\_FOREIGN\_KEYS=true

**Microsoft SQL Server**

Đối với cấu hình Microsoft SQL Server, cần phải đảm bảo đã cài đặt các extension sqlsrv và pdo\_sqlsrv cho PHP.

**Cấu hình sử dụng URL**

Thông thường, các cấu hình đến cơ sở dữ liệu sẽ có các thông tin như host, database, username, password… Tuy nhiên, một số nhà cung cấp như AWS, Heroku sử dụng một đường dẫn URL để chứa tất cả thông tin cấu hình đến cơ sở dữ liệu. Do đó, Laravel cung cấp thêm biến môi trường DATABASE\_URL trong .env. URL này sẽ có dạng:

driver://username:password@host:port/database?options

Ví dụ: Kết nối database forge qua mysql với username là root, password là password với host 127.0.0.1 có options là charset=UTF-8:

mysql://root:password@127.0.0.1/forge?charset=UTF-8

## Kết nối Read-Write

Trong trường hợp cần sử dụng các kết nối riêng biệt cho các câu lệnh SELECT và các câu lệnh INSERT, UPDATE và DELETE, ta có thể thiết lập cấu hình như sau:

'mysql' => [

'read' => [

'host' => [

'192.168.1.1',

'196.168.1.2',

],

],

'write' => [

'host' => [

'196.168.1.3',

],

],

'sticky' => true,

'driver' => 'mysql',

'database' => 'database',

'username' => 'root',

'password' => '',

'charset' => 'utf8mb4',

'collation' => 'utf8mb4\_unicode\_ci',

'prefix' => '',

],

## Thực thi truy vấn

Sau khi đã kết nối, ta có thể thực thi các truy vấn đến database thông qua lớp DB kế thừa \Illuminate\Support\Facades\DB.

**SELECT**

Để chạy một truy vấn SELECT cơ bản, ta dùng phương thức select.

use Illuminate\Support\Facades\DB;

$users = DB::select("SELECT \* FROM users where active = ?", [1]);

foreach ($users as $user) {

echo $user->name;

}

Trong đó, tham số đầu tiên là câu truy vấn SQL, còn tham số thứ hai là ràng buộc dữ liệu. Thông thường, đây là các giá trị của các điều kiện trong mệnh đề where. Giá trị ràng buộc nhằm cung cấp bảo vệ dữ liệu khỏi SQL injection.

Phương thức select sẽ trả về một array, trong đó mỗi một kết quả trong array sẽ là một stdClass. Ngoài ra, ta cũng có thể thêm ràng buộc dữ liệu vào câu truy vấn bằng cách gán tên cho nó:

$users = DB::select('select \* from users where id = :id',

['id' => 1]);

**INSERT**

Để thêm dữ liệu vào database, sử dụng phương thức insert:

use Illuminate\Support\Facades\DB;

DB::insert("insert into users (id, name)

values (1, 'Marc')");

Ràng buộc truy vấn trong phương thức insert:

use Illuminate\Support\Facades\DB;

DB::insert('insert into users (id, name) values (?, ?)', [1, 'Marc']);

**UPDATE**

Để cập nhật dữ liệu trong database, ta sử dụng phương thức update:

use Illuminate\Support\Facades\DB;

$affected = DB::update("update users set votes = 100

where name = 'Anita'");

Tương tự, select và insert, ta cũng có thể ràng buộc truy vấn trong phương thức update:

use Illuminate\Support\Facades\DB;

$affected = DB::update('update users set votes = ? where name = ?', [100, 'Anita']);

**DELETE**

Để xóa dữ liệu khỏi database, sử dụng phương thức delete:

use Illuminate\Support\Facades\DB;

$deleted = DB::delete('delete from users');

**Thực thi các câu lệnh chung**

Trong trường hợp sử dụng một số cấu lệnh không trả về bất cứ giá trị nào hoặc không muốn nhận dữ liệu trả về, ta sử dụng phương thức statement:

DB::statement('drop table users');

**Chỉ định kết nối cơ sở dữ liệu**

Khi cần chỉ định kết nối sẽ thực thi truy vấn đến cơ sở dữ liệu, ta sử dụng phương thức connection trước khi thực thi. Tên kết nối trong connection phải tương ứng với một trong các kết nối được liệt kê trong file cấu hình config/database.php.

Ví dụ: Chỉ định kết nối đến sqlite.

use Illuminate\Support\Facades\DB;

$users = DB::connection('sqlite')->select(...);

Ngoài ra, ta cũng có thể truy cập PDO raw cơ bản của một kết nối bằng cách sử dụng phương thức getPdo:

$pdo = DB::connection()->getPdo();

## Listen Query Event.

Để lắng nghe các sự kiện truy vấn, ta sử dụng phương thức listen và Laravel khuvến khích nên đặt chúng vào phương thức boot của một Service Provider:

<?php

namespace App\Providers;

use Illuminate\Support\Facades\DB;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider

{

public function register()

{

//

}

public function boot()

{

DB::listen(function ($query) {

// $query->sql

// $query->bindings

// $query->time

});

}

}

## Database transaction

Để thực thi nhiều câu truy vấn trong một giao dịch của cơ sở dữ liệu, ta sử dụng phương thức transaction của facade DB. Nếu một ngoại lệ được ném ra trong quá trình thực thi, transaction sẽ tự động được khôi phục lại. Nếu thực thi thành công, transaction sẽ tự động được commit. Ta không cần phải lo lắng về việc rollback hoặc commit theo cách thủ công khi sử dụng phương thức này.

use Illuminate\Support\Facades\DB;

DB::transaction(function () {

DB::update('update users set votes = 1');

DB::delete('delete from posts');

});

**Xử lí bế tắc**

Ta có thể xác định số lần thử lại transaction bằng cách truyền số lần thực thi lại transaction vào tham số thứ hai trong phương thức transaction. Nếu hết số lần thực thi lại mà transaction vẫn chưa hoàn thành, Laravel sẽ ném ra một ngoại lệ. Ví dụ, thử lại transaction 5 lần:

use Illuminate\Support\Facades\DB;

DB::transaction(function () {

DB::update('update users set votes = 1');

DB::delete('delete from posts');

}, 5); 🡸

Ngoài ra, ta cũng có thể thực thi transaction một cách thủ công qua các phương thức như beginTransaction, rollback, commit. Ví dụ:

try {

DB::beginTransaction();

DB::update('update users set votes = 1');

DB::delete('delete from posts');

DB::commit();

}

catch (Exception $exception) {

DB::rollBack();

}

## Kết nối đến database CLI.

Để kết nối đến database qua CLI, sử dụng câu lệnh command:

php artisan db

Câu lệnh trên sẽ sử dụng kết nối mặc định. Nếu cần chỉ định kết nối database ta có thể truyền tên kết nối vào command. Ví dụ, kết nối đến mysql:

php artisan db mysql

## Migration và Seeder

**Migration**

Migration trong Laravel giúp ta kiểm soát các phiên bản của cơ sở dữ liệu và hỗ trợ thêm, sửa, xóa và các tác vụ quản lý cơ sở dữ liệu khi làm việc, đặc biệt là làm việc theo nhóm. Nó cho phép mỗi thành viên trong nhóm xác định và chia sẻ định nghĩa lược đồ cơ sở dữ liệu của ứng dụng một cách tự động mà không cần phải đối mặt với các vấn đề do việc di chuyển cơ sở dữ liệu gây ra.

Các file migration được tạo sẽ lưu trong thư mục database/migrations và kế thừa lớp Migration. Để tạo một migration cho bảng flights, sử dụng câu lệnh:

php artisan make:migration create\_flights\_model

Kết quả:

<?php

use Illuminate\Database\Migrations\Migration;

use Illuminate\Database\Schema\Blueprint;

use Illuminate\Support\Facades\Schema;

class CreateFlightsTable extends Migration

{

public function up()

{

Schema::create('flights', function (Blueprint $table) {

$table->id();

$table->string('name');

$table->string('airline');

$table->timestamps();

});

}

public function down()

{

Schema::drop('flights');

}

}

Trong đó, phương thức up chứa dữ liệu để tạo các trường của bảng kèm kiểu dữ liệu. Phương thức down dùng để xóa bảng. Để chạy migration, sử dụng câu lệnh:

php artisan migrate

Trong trường hợp muốn quay lại, sử dụng câu lệnh rollback. Ta có thể thêm flag --step để chọn số bước nhảy mong muốn.

php artisan migrate:rollback

php artisan migrate:rollback –step=5

**Seeder**

Bên cạnh việc kiểm soát các phiên bản cơ sở dữ liệu, Laravel cung cấp thêm chức năng hỗ trợ ta “gieo” dữ liệu vào cơ sở dữ liệu. Tất cả các lớp hạt (seed) sẽ được lưu trữ trong thư mục database/seeders. Để tạo một seeder, sử dụng câu lệnh artisan:

php artisan make:seeder UserSeeder

Một file seeder sẽ được tạo ra với nội dung như sau:

<?php

namespace Database\Seeders;

use Illuminate\Database\Seeder;

use Illuminate\Support\Facades\DB;

use Illuminate\Support\Facades\Hash;

use Illuminate\Support\Str;

class DatabaseSeeder extends Seeder

{

public function run()

{

DB::table('users')->insert([

'name' => Str::random(10),

'email' => Str::random(10).'@gmail.com',

'password' => Hash::make('password'),

]);

}

}

Ta cũng có thể dùng model factory để tạo dữ liệu seeder. Ví dụ, tạo 50 người dùng có bài post liên quan:

use App\Models\User;

public function run()

{

User::factory()

->count(50)

->hasPosts(1)

->create();

}

Phương thức call kế thừa từ lớp DatabaseSeeder cho phép ta kiểm soát thứ tự dữ liệu “gieo”. Ví dụ: chỉ “gieo” dữ liệu của UserSeeder, PostSeeder và CommentSeeder:

public function run()

{

$this->call([

UserSeeder::class,

PostSeeder::class,

CommentSeeder::class,

]);

}

Để tiến hành gieo, sử dụng câu lệnh artisan sau:

# Chạy toàn bộ seeder

php artisan db:seed

# Chỉ gieo dữ liệu trong UserSeeder

php artisan db:seed --class=UserSeeder

# Eloquent ORM

Eloquent ORM là một trình ánh xạ quan hệ đối tượng (Object-Relational Mapper) giúp cho việc tương tác với cơ sở dữ liệu trở nên thú vị và tiện lợi hơn. Với Eloquent ORM, mỗi bảng trong cơ sở dữ liệu ánh xạ một "Model" tương ứng để có thể tương tác với bảng đó. Bên cạnh việc cho phép truy xuất các bản ghi (record) từ bảng, Eloquent ORM còn cho phép thêm, cập nhật và xóa các bản ghi ra khỏi bảng trong cơ sở dữ liệu một cách dễ dàng. So với QueryBuilder, Eloquent ORM có tính bảo mật cao hơn và tốt hơn trong việc phòng chống SQL Injection.

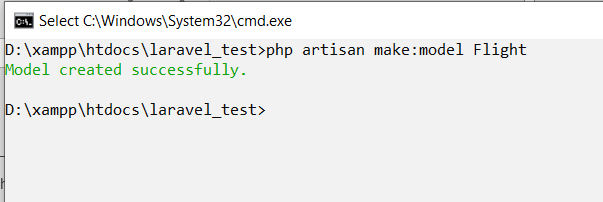
## Tạo Eloquent Model

Từ Laravel 8, tất cả model được tạo ra sẽ nằm trong thư mục app\Models. Các model đều phải kế thừa lớp Illuminate\Database\Eloquent\Model.

Để tạo một model, từ command line ta sử dụng cú pháp sau với ModelName là tên model cần tạo:

php artisan make:model ModelName

Ví dụ, tạo model Flight:



Hình 9.1: Tạo Model

Thư mục app\Models có thêm file Flight.php với nội dung như sau:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

use HasFactory;

}

Ta có thể tạo đồng thời model và migration cho table bằng cách thêm flag --migration hoặc -m.

php artisan make:model Flight --migration

php artisan make:model Flight -m

Ngoài migration, ta cũng có thể tạo đồng thời nhiều loại lớp khác như seeder, factory, controller cho model:

# Tạo model Flight kèm FlightFactory...

php artisan make:model Flight --factory

php artisan make:model Flight -f

# Tạo model Flight kèm FlightSeeder...

php artisan make:model Flight --seed

php artisan make:model Flight -s

# Tạo model Flight kèm FlightController...

php artisan make:model Flight --controller

php artisan make:model Flight -c

Laravel có hỗ trợ gộp các tùy chọn vào chung một câu lệnh:

# Tạo model kèm migration, factory, seeder, và controller...

php artisan make:model Flight -mfsc

# Hoặc

php artisan make:model Flight --all

# Tạo model kèm các pivot...

php artisan make:model Member --pivots

## Cấu hình Eloquent Model

**Tên bảng**

Mặc định, Laravel quy ước tên bảng trong model được đặt theo tên của model viết dưới dạng snake\_case và thể hiện ở dạng số nhiều. Ví dụ, model Flight vừa tạo có tên table là flights, hoặc model AirTrafficController thì sẽ có bảng là air\_traffic\_controllers.

Trong trường hợp muốn đặt tên cho bảng khác với quy ước, ta có thể chỉnh lại bằng cách gán tên mới vào thuộc tính $table. Ví dụ, định nghĩa tên table của model Flight là my\_flights:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

protected $table = 'my\_flights'; 🡸

}

**Primary Key**

Eloquent mặc định id sẽ là khóa chính trong model. Để thay đổi khóa chính, ta định nghĩa lại bằng thuộc tính $primaryKey. Ví dụ, định nghĩa khóa chính của model Flight thành flight\_id:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

protected $primaryKey = 'flight\_id';

}

Ngoài ra, Eloquent cũng mặc định khóa chính sẽ có kiểu dữ liệu tự động tăng (increment) và sẽ tự động được ép thành kiểu số. Nếu không muốn khóa chính là increment hoặc không muốn ép khóa chính về kiểu số, ta có thể gán false cho $incrementing:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

public $incrementing = false; 🡸

}

Trong trường hợp muốn xác định lại kiểu dữ liệu của khóa chính, ta định nghĩa lại bên trong thuộc tính $keyType.Ví dụ, thiết lập khóa chính có kiểu dữ liệu string:

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

protected $keyType = 'string'; 🡸

}

**Timestamp**

Mặc định, Eloquent sẽ thêm 2 cột created\_at và updated\_at vào bảng và sẽ tự động gán giá trị cho 2 cột này cho các trường hợp tạo mới, cập nhật dữ liệu trong bảng. Gán false cho $timestamps để tắt tự động gán giá trị cho 2 cột này:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

public $timestamps = false; 🡸

}

Để đổi tên hai cột này mà vẫn lưu giá trị vào cơ sở dữ liệu, ta làm theo cách sau:

<?php

class Flight extends Model

{

const CREATED\_AT = 'creation\_date';

const UPDATED\_AT = 'updated\_date';

}

**Kết nối cơ sở dữ liệu**

Eloquent sử dụng kết nối mặc định được thiết lập trong ứng dụng. Trong hợp model cần sử dụng một connection khác, ta thay đổi thiết lập bằng thuộc tính $connection. Ví dụ, model Flight chuyển kết nối sang sqlite:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

protected $connection = 'sqlite'; 🡸

}

**Giá trị thuộc tính mặc định**

Mặc định, một instance của một model sẽ không chứa bất kì giá trị của một thuộc tính (attribute) nào. Để gán các giá trị mặc định cho thuộc tính nào đó, ta cần thiết lập trong $attributes. Trong đó, attribute là các cột trong table của cơ sở dữ liệu:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model

{

protected $attributes = [

'delayed' => false, 🡨

];

}

## Truy vấn cơ bản

**Query**

Khi tạo xong một model và bảng liên quan, ta đã có thể truy xuất dữ liệu từ cơ sở dữ liệu. Có thể xem mỗi eloquent model như một trình tạo truy vấn (query builder) mạnh mẽ, cho phép ta truy vấn dữ liệu của table được liên kết với model đó. Để truy xuất tất cả dữ liệu của table đã liên kết với model, sử dụng phương thức all:

use App\Models\Flight;

foreach (Flight::all() as $flight) {

echo $flight->name;

}

Vì mỗi eloquent model đóng vai trò như một trình tạo truy vấn, ta hoàn toàn có thể thêm các ràng buộc để truy vấn dữ liệu và sử dụng phương thức get để lấy dữ liệu trả về:

$flights = Flight::where('active', 1)

->orderBy('name')

->take(10)

->get();

Nếu eloquent model đã được truy vấn, ta có thể làm mới dữ liệu bằng cách sử dụng phương thức fresh.

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

$freshFlight = $flight->fresh();

Trong trường hợp cần làm mới cả dữ liệu truy vẫn lẫn dữ liệu của các mối quan hệ khác, ta sử dụng phương thức refresh.

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

$flight->number = 'FR 456';

$flight->refresh(); 🡸

$flight->number; // "FR 900"

**Collection**

Khi truy xuất nhiều record từ cơ sở dữ liệu bằng phương thức all và get, dữ liệu sẽ không được trả về bằng một mảng PHP thuần. Eloquent sẽ trả về một collection thuộc lớp Illuminate\Database\Eloquent\Collection.

Lớp Collection kế thừa Illuminate\Support\Collection, lớp này cung cấp nhiều phương thức hữu ích để tương tác với các collection data. Ví dụ, phương thức reject được sử dụng để xóa các model khỏi collection:

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

$flights = $flights->reject(function ($flight) {

return $flight->cancelled;

});

Ngoài các phương thức được cung cấp bởi lớp base collection của Laravel, lớp Eloquent collection cũng cung cấp một số phương thức bổ sung dành riêng cho việc tương tác với collection của các Eloquent model.

Vì tất cả các collection của Laravel đều triển khai (implement) các giao diện lặp lại (iterable interfaces) của PHP nên ta có thể lặp các collection như một mảng:

foreach ($flights as $flight) {

echo $flight->name;

}

**Chunk data**

Khi cần tải một lượng lớn dữ liệu, bộ nhớ của ứng dụng có thể bị thiếu nếu sử dụng phương thức all và get. Eloquent cung cấp phương thức chunk để giải quyết vấn đề này. chunk sẽ chia dữ liệu thành các mảnh nhỏ để tiện cho việc xử lí và tránh hết dung lượng bộ nhớ.

use App\Models\Flight;

Flight::chunk(200, function ($flights) {

foreach ($flights as $flight) {

//

}

});

Để chia mảnh dữ liệu theo cột id, ta có thể sử dụng phương thức chunkById.

Flight::where('departed', true)

->chunkById(200, function ($flights) {

$flights->each->update(['departed' => false]);

}, $column = 'id');

## Truy vấn nâng cao

Eloquent cho phép lồng các truy vấn con (subquery), hỗ trợ ta có thể truy xuất dữ liệu bằng một câu truy vấn duy nhất.

**select subquery**

Để lồng một truy vấn khác vào câu lệnh select, ta truyền thêm một builder vào trong phương thức select hoặc addSelect.

Ví dụ, ta có bảng destinations chứa các điểm đến và bảng flight chứa các chuyến bay. Để xuất các điểm đến và tên các chuyến bay đã đến điểm đến đó, ta dùng câu truy vấn sau:

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();

**oderBy subquery**

Tương tự ta cũng có thể sắp xếp bằng oderBy. Ví dụ, sắp xếp các điểm đến theo thời gian đón chuyến bay cuối cùng theo thứ tự giảm dần:

return Destination::orderByDesc(

Flight::select('arrived\_at')

->whereColumn('destination\_id', 'destinations.id')

->orderByDesc('arrived\_at')

->limit(1)

)->get();

**Truy xuất đơn lẻ, tổng hợp**

Nếu chỉ cần trả về kết quả đầu tiên, ta sử dụng phương thức find, first hoặc firstWhere:

use App\Models\Flight;

// Xuất dữ liệu theo primary key...

$flight = Flight::find(1);

// Xuất dữ liệu đầu tiên

$flight = Flight::where('active', 1)->first();

// Thay thế để truy xuất dữ liệu đầu tiên phù hợp với các ràng buộc truy vấn ...

$flight = Flight::firstWhere('active', 1);

Phương thức firstOr sẽ trả về kết quả phù hợp đầu tiên của truy vấn hoặc thực thi function bên trong:

$model = Flight::where('legs','>', 3)->firstOr(

function () {

// ...

});

Hai phương thức firstOrCreate và firstOrNew sẽ tạo mới nếu không tìm thấy dữ liệu nào.

use App\Models\Flight;

// Xuất chuyến bay theo name hoặc tạo mới nếu không tồn tại

$flight = Flight::firstOrCreate([

'name' => 'London to Paris'

]);

// Xuất chuyến bay theo name hoặc tạo mới theo name, delayed, arrival\_time

$flight = Flight::firstOrCreate(

['name' => 'London to Paris'],

['delayed' => 1, 'arrival\_time' => '11:30']

);

// Xuất chuyến bay theo name hoặc khởi tạo instance Chuyến bay mới

$flight = Flight::firstOrNew([

'name' => 'London to Paris'

]);

// Xuất chuyến bay theo name hoặc khởi tạo instance Chuyến bay mới theo name, delayed, and arrival\_time attributes...

$flight = Flight::firstOrNew(

['name' => 'Tokyo to Sydney'],

['delayed' => 1, 'arrival\_time' => '11:30']

);

Khi tương tác với các eloquent model, ta cũng có thể sử dụng các hàm như count, sum, max, và các phương thức tổng hợp được cung cấp bởi query builder. Các phương thức này trả về một giá trị vô hướng thay vì tạo một instance cho eloquent model:

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

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

**Chèn dữ liệu**

Bên cạnh việc truy xuất, ta cũng cần phải chèn thêm các record mới. Để tạo một record mới vào cơ sở dữ liệu với Eloquent, nên khởi tạo một instance mới cho model và thiết lập các thuộc tính cho nó. Sau đó, gọi phương thức save để lưu lại. Ví dụ:

<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;

use App\Models\Flight; 🡸

use Illuminate\Http\Request;

class FlightController extends Controller

{

//Tạo một chuyến bay mới với dữ liệu từ $request

public function store(Request $request)

{

// Kiểm tra dữ liệu trong $request

$flight = new Flight;

$flight->name = $request->name; 🡸

$flight->save();

}

}

Trong ví dụ trên, ta gán name từ HTTP request cho thuộc tính name của App\Models\Flight. Khi gọi phương thức save, một record sẽ được thêm vào cơ sở dữ liệu. Dữ liệu created\_at và updated\_at sẽ tự động được đắt phương thức save được gọi.

Eloquent cũng cung cấp phương thức create để "lưu" một model mới bằng cách sử dụng một câu lệnh PHP. Instance của model được chèn sẽ được trả lại cho bạn theo createphương thức:

use App\Models\Flight;

$flight = Flight::create([

'name' => 'London to Paris',

]);

Tuy nhiên, ta cần chỉ định một fillable hoặc thuộc tính guarded trong lớp của model đó trước khi sử dụng phương thức create. Các thuộc tính này là bắt buộc vì tất cả các eloquent model đều được bảo vệ khỏi các lỗ hổng phân công hàng loạt (mass assignment) theo mặc định.

**Cập nhật dữ liệu**

Phương thức save cũng có thể được sử dụng để model cập nhật record đã tồn tại trong cơ sở dữ liệu. Để cập nhật một model, ta nên truy xuất model đó rồi cập nhật dữ liệu và gọi phương thức save để lưu lại:

use App\Models\Flight;

$flight = Flight::find(1);

$flight->name = 'Paris to London';

$flight->save();

**Cập nhật hàng loạt**

Ta có thể cập nhật nhiều model bằng một câu truy vấn. Ví dụ, tất cả các chuyến bay đang hoạt động có điểm đến là San Diego sẽ được đánh dấu là bị trì hoãn:

Flight::where('active', 1)

->where('destination', 'San Diego')

->update(['delayed' => 1]);

Phương thức update chứa một mảng các cặp cột và giá trị cần được cập nhật. Khi thực hiện cập nhật hàng loạt qua Eloquent, các sự kiện như saving, saved, updating, và updated cho các model cần cập nhật. Do các model không bao giờ thực sự được truy xuất khi thực hiện cập nhật hàng loạt.

**Kiểm tra các thay đổi của thuộc tính**

Eloquent cung cấp các phương thức isDirty, isClean và wasChanged để kiểm tra trạng thái của model và xác định cách thuộc tính của nó có thay đổi từ khi model được truy xuất.

Phương thức isDirty kiểm tra dữ liệu trong model đã từng thay đổi chưa, còn isClean sẽ kiểm tra dữ liệu trong model đã được dọn dẹp hay chưa.

use App\Models\User;

$user = User::create([

'first\_name' => 'Taylor',

'last\_name' => 'Otwell',

'title' => 'Developer',

]);

$user->title = 'Painter';

// Kiểm tra dữ liệu trong model đã thay đổi chưa?

$user->isDirty(); // true

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

$user->isDirty('first\_name'); // false

// Kiểm tra dữ liệu trong model đã được làm sạch?

$user->isClean(); // false

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

$user->isClean('first\_name'); // true

$user->save();

$user->isDirty(); // false

$user->isClean(); // true

Phương thức wasChanged dùng để xác định nếu có các thuộc tính đã được thay đổi khi mô hình được lưu lần cuối trong chu kì request hiện tại:

$user = User::create([

'first\_name' => 'Taylor',

'last\_name' => 'Otwell',

'title' => 'Developer',

]);

$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 trước khi bị thay đổi.

$user = User::find(1);

$user->name; // John

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

$user->name = "Jack";

$user->name; // Jack

$user->getOriginal('name'); // John

$user->getOriginal(); // Mảng với các thuộc tính ban đầu

**Xóa dữ liệu**

Để xóa dữ liệu một model, ta sử dụng phương thức delete:

use App\Models\Flight;

$flight = Flight::find(1);

$flight->delete();

Trong trường hợp cần xóa tất cả các record liên quan đến model đó, Eloquent cung cấp phương thức truncate. truncate cũng sẽ thiết lập lại ID tự động tăng trên bất kì các bảng liên quan với model đó:

Flight::truncate();

Ngoài ra, ta có thể sử dụng phương thức destroy khi biết khóa chinh của model cần xóa. Phương thức này có thể xóa một hoặc nhiều khóa chính cũng như xóa theo mảng, theo collection:

Flight::destroy(1);

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

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

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

Ta cũng có thể tạo một truy vấn Eloquent để xóa tất cả các model theo yêu cầu. Ví dụ, xóa tất cả các chuyến bay được đánh dấu là không hoạt động.

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

Tương tự cập nhật hàng loạt, khi thực hiện câu lệnh xóa hàng loạt qua Eloquent thì các sự kiện deleting và deleted sẽ không được gửi cho các model đã xóa.

**Xóa mềm**

Eloquent cũng hỗ trợ "soft deleting" các model. Vói soft deleting, dữ liệu trong model sẽ không bị xóa mà sẽ có thêm thuộc tính deleted\_at được thêm vào model để ghi lại thời gian model đó bị "xóa".

Thêm lớp Illuminate\Database\Eloquent\SoftDeletes vào model để sử dụng tính năng này:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

use Illuminate\Database\Eloquent\SoftDeletes; 🡸

class Flight extends Model

{

use SoftDeletes;

}

# Relationship trong Eloquent ORM

Các bảng trong cơ sở dữ liệu thường có các mối quan hệ (relationship) với nhau, Eloquent cũng hỗ trợ tạo các mối quan hệ giữa các model. Từ đó giúp việc quản lí và làm việc với các relationship một cách dễ dàng và đơn giản. Ví dụ: một bài viết có thể có nhiều bình luận hoặc một mã đơn hàng chỉ có một người đặt.

Các eloquent relationship trong model được định nghĩ tương tự như các phương thức bình thường. Vì thế, ta cũng có thể dùng chúng để tạo các query builder.

## One to one

Mối quan hệ “một-một” (1-1) là một trong những mối quan hệ đơn giản nhất trong cơ sở dữ liệu. Để định nghĩa mối quan hệ này, ta dùng phương thức hasOne với cú pháp:

hasOne($relationModel, $foreignKey, $localKey');

Trong đó:

* $relationModel: model sẽ được liên kết với model hiện tại theo mối quan hệ 1-1.
* $foreignKey: khóa ngoại của table $relationModel để liên kết hai bảng.
* $localKey: cột chứa dữ liệu để liên kết với table của $relationModel. $localKey thường là khóa chính của model hiện tại.

Nếu không truyền $foreignKey và $localKey, model sẽ ngầm định theo mặc định. Ví dụ, mỗi người dùng chỉ có duy nhất một số điện thoại. $foreignKey là user\_id và $localKey là id:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

// Liên kết số điện thoại với người dùng

public function phone()

{

return $this->hasOne(Phone::class);

}

}

Sau khi thiết lập mối quan hệ, ta có thể xuất dữ liệu trong model liên quan.

// xuất dữ liệu trong table phone của user id = 1

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

**Đảo ngược mối quan hệ 1-1**

Do một người dùng có một số điện thoại nên ta cũng có mối quan hệ đảo ngược là một số điện thoại chỉ thuộc về một người dùng. Để định nghĩa mối quan hệ đảo ngược của quan hệ 1-1, ta sủ dụng phương thức belongTo:

belongsTo($relatedModel, $foreignKey, $ownerKey);

Ví dụ, định nghĩa mối quan hệ đảo ngược cho model Phone và User:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model

{

// Một số điện thoại chỉ thuộc về một người dùng

public function user()

{

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

}

}

Truy vấn User theo Phone:

// Lấy người dùng có số điện thoại +84123456780

$user = Phone::where('number', '+84123456780')->user;

## One to many

Mối quan hệ “một-nhiều” (1-n) được sử dụng trong trường hợp một model sẽ được liên kết đến một hoặc nhiều model khác. Để định nghĩa mối quan hệ này chúng ta sử dụng phương thức hasMany với cú pháp tương tự hasOne.

Ví dụ: Định nghĩa quan hệ 1-n giữa model Post và model Comment.

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model

{

// Lấy các bình luận của bài Post

public function comments()

{

return $this->hasMany(Comment::class);

}

}

Với mối quan hệ 1-n, dữ liệu trả về là một collection với các item sẽ là các model. Ví dụ, lấy các comment trong Post 1:

use App\Models\Post;

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

foreach ($comments as $comment) {

//

}

Ta cũng có thể thêm ràng buộc cho các query builder. Ví dụ, lấy các comment của Post có title là foo:

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

->where('title', 'foo')

->first();

**Đảo ngược mối quan hệ 1-n**

Tương tự mối quan hệ 1-1, ta cũng có thể định nghĩa mối quan hệ đảo ngược cho các mối quan hệ 1-n. Ở ví dụ trên, ta có thể thấy một comment chỉ thuộc về một post nên model Comment và Post sẽ có mối quan hệ belongTo.

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model

{

// Lấy bài post của một bình luận

public function post()

{

return $this->belongsTo(Post::class);

}

}

Query builder lấy comment có post là title:

use App\Models\Comment;

$comment = Comment::find(1);

return $comment->post->title;

## Has one of many

Giả sử trong môt cơ sở dữ liệu bán hàng, một người dùng có thể có nhiều đơn hàng. Ta có thể dễ dàng lấy đơn hàng gần nhất mà người dùng đã đặt bằng cách sử dụng hasOne kết hợp với phương thức lasterOfMany:

// Lấy thông tin đơn hàng gần nhất của người dùng

public function latestOrder()

{

return $this->hasOne(Order::class)->latestOfMany();

}

Tương tự, ta có thể dùng phương thức oldestOfMany để truy xuất model đầu tiên hoặc cũ nhất liên quan:

// Lấy thông tin đơn hàng đầu tiên của người dùng

public function oldestOrder()

{

return $this->hasOne(Order::class)->oldestOfMany();

}

Theo mặc định, các phương thức latestOfMany và oldestOfMany sẽ truy xuất model liên quan mới nhất hoặc cũ nhất dựa trên khóa chính của model đồng thời dữ liệu cũng có thể được sắp xếp theo khóa chính. Tất nhiên, ta có thể tùy biến lại bằng phương thức ofMany. Phương thức này nhận vào tên cột cần sắp xếp và giá trị sắp xếp theo min hoặc max để áp dụng khi truy vấn. Ví dụ, lấy thông tin đơn hàng đắt nhất của người dùng:

// Lấy thông tin đơn hàng đắt nhất của người dùng

public function largestOrder()

{

return $this->hasOne(Order::class)->ofMany('price', 'max');

}

## Has one thourgh

Mối quan hệ "có-một-thông qua" thực chất cũng là mối quan hệ 1-1 tuy nhiên chúng được nối với nhau thông qua một model khác. Ví dụ, một bộ phận là của một chiếc xe và một chiếc xe đó chỉ thuộc một chủ sở hữu. Để kiểm tra xem một bộ phận của chiếc xe thuộc về ai, ta cần biết chiếc xe đó do ai sở hữu. Chiếc xe chính là trung gian. Ta có các bảng sau:

mechanics

id - integer

name – string

cars

id - integer

model - string

mechanic\_id - integer

owners

id - integer

name - string

car\_id - integer

Để định nghĩa ai là owner của mechanics, ta sử dụng phương thức hasOneThrough:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Mechanic extends Model

{

// Lấy thông tin chủ xe

public function carOwner()

{

return $this->hasOneThrough(Owner::class, Car::class);

}

}

Trong trường hợp tên cột không theo rule, ta có thể tùy chỉnh trong phương thức hasOneThrough:

class Mechanic extends Model

{

// Lấy thông tin chủ xe

public function carOwner()

{

return $this->hasOneThrough(

Owner::class,

Car::class,

'mechanic\_id', // Khóa ngoại bảng cars

'car\_id', // Khóa ngoại bảng cars owners

'id', // Local key trong bảng mechanics

'id' // Local key trong bảng cars

);

}

}

## Has many thourgh

Mối quan hệ "có-nhiều-thông qua" tương tự has one thourgh nhưng nó cần hơn một model thứ ba để xác định. Ví dụ: Một dự án có thể có nhiều môi trường và một môi trường có thể có nhiều lần triển khai. Môi trường chính là trung gian. Ta có các bảng sau:

projects

id - integer

name - string

environments

id - integer

project\_id - integer

name - string

deployments

id - integer

environment\_id - integer

commit\_hash - string

Để định nghĩa tất cả deployments của mechanics, ta sử dụng phương thức hasManyThrough:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Project extends Model

{ // Lấy tất cả các thông tin triển khai của dự án

public function deployments()

{

return $this->hasManyThrough(Deployment::class,

Environment::class);

}

}

Trong trường hợp tên cột không theo rule, ta cũng có thể tùy chỉnh trong phương thức hasManyThrough:

class Project extends Model

{

public function deployments()

{

return $this->hasManyThrough(

Deployment::class,

Environment::class,

'project\_id', // Khóa ngoại bảng environments

'environment\_id', // Khóa ngoại bảng deployments

'id', // Local key trong bảng projects

'id' // Local key trong bảng environments

);

}

}

## Many to many

Mối quan hệ “nhiều-nhiều” (n-n) phức tạp hơn một chút so với mối quan hệ 1-1 và 1-n. Ta thường gặp mối quan hệ này trong phân quyền ứng dụng, một người dùng có thể có nhiều vai trò và những người dùng khác cũng có những vai trò đó.

Ví dụ, một người dùng có thể là "Tác giả" và "Biên tập viên" của một ứng dụng và ứng dụng đó lại có nhiều “Tác giả” và nhiều “Biên tập viên”. Do đó, một người dùng sẽ có nhiều vai trò và một vai trò cũng có nhiều người dùng.

users

id - integer

name - string

roles

id - integer

name - string

role\_user

user\_id - integer

role\_id – integer

Ta có thể thấy có hai bảng users và roles chứa thông tin các người dùng và các vai trò, để lấy ra được các vai trò của một người dùng thì ta cần thêm bảng role\_user để biết user\_id đó có những role\_id nào.

Định nghĩa quan hệ n-n, ta sử dụng phương thức belongsToMany.

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model

{

//Các vai trò của người dùng

public function roles()

{

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

}

}

Tương tự, ta có thể định nghĩa mối quan hệ đảo ngược để biết những người dùng nào có cùng một vai trò:

<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Role extends Model

{

// Những người dùng có cùng một vai trò

public function users()

{

return $this->belongsToMany(User::class);

}

}

**Bảng trung gian**

Để lấy thêm dữ liệu của bảng trung gian, sử dụng thuộc tính pivot. Ví dụ, lấy created\_at của user\_role:

use App\Models\User;

$user = User::find(1);

foreach ($user->roles as $role) {

echo $role->pivot->created\_at;

}

Giả sử, những người dùng trong ứng dụng có thể đăng ký nhiều podcast, ta sẽ có mối quan hệ n-n giữa người dùng và podcast. Ta có thể đổi tên thuộc tính pivot của bảng trung gian thành subscription bằng cách sử dụng phương thức as trong khi xác định mối quan hệ:

return $this->belongsToMany(Podcast::class)

->as('subscription') 🡸

->withTimestamps();

Từ đây, ta đã có thể truy cập dữ liệu của bảng trung gian bằng tên mới, giúp code dễ hiểu và liên kết hơn:

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

foreach ($users->flatMap->podcasts as $podcast) {

echo $podcast->subscription->created\_at; 🡸

}

## Default Model

Các phương thức belongsTo, hasOne, hasOneThrough và morphOne mặc định sẽ trả về null nếu không tồn tại dữ liệu của quan hệ trong cơ sở dữ liệu. Trong trường hợp muốn thay null thành một đối tượng model không có giá thuộc tính, ta sử dụng phương thức withDefault.

// Lấy tác giả của bài viết

public function user()

{

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

}

Để gán thêm các thuộc tính mặc định truyền vào model, ta truyền thêm tham số là một mảng vào phương thức withDefault:

// Lấy tác giả của bài viết

public function user()

{

return $this->belongsTo(User::class)->withDefault([

'name' => 'Guest Author', 🡸

]);

}

Hoặc sử dụng một closure (bao đóng) trả về mảng giá trị của thuộc tính:

// Lấy tác giả của bài viết

public function user()

{

return $this->belongsTo(User::class)

🡺 ->withDefault(function ($user, $post) {

$user->name = 'Guest Author';

});

}