Course - Laravel Framework

Artisan Console

Artisan là giao diện command line (cli) đi kèm với Laravel. Artisan tồn tại ở gốc ứng dụng của bạn dưới dạng artisan script thủ công và cung cấp một số lệnh hữu ích có thể hỗ trợ bạn trong khi xây dựng ứng dụng của mình.

Tags: laravel artisan console, laravel framework

Giới thiệu

Artisan được đặt ở thư mục gốc của ứng dụng dưới dạng artisan script thủ công và cung cấp một số lệnh hữu ích có thể hỗ trợ bạn trong khi xây dựng ứng dụng của mình. Để xem danh sách tất cả các lệnh Artisan có sẵn, bạn có thể sử dụng lệnh `list`:

```
php artisan list
```

Mọi lệnh đều có một màn hình "help" sẽ hiển thị và mô tả các đối số và tùy chọn có sẵn của lệnh. Để xem màn hình help, hãy đặt trước tên của lệnh với option `help`, ví dụ:

```
php artisan help migrate
```

Laravel Sail

Nếu bạn đang sử dụng Laravel Sail làm môi trường phát triển local của mình, hãy nhớ sử dụng dòng lệnh `sail` để gọi các lệnh Artisan. Sail sẽ thực hiện các lệnh Artisan của bạn trong các container của Docker của ứng dụng của bạn:

```
./sail artisan list
```

Tinker (REPL)

Laravel Tinker là một công cụ REPL mạnh mẽ cho framework Laravel, được cung cấp bởi package PsySH.

Cài đặt

Tất cả các ứng dụng Laravel đều có sẵn Tinker theo mặc định. Tuy nhiên, bạn có thể cài đặt Tinker bằng Composer nếu trước đó bạn đã xóa nó khỏi ứng dụng của mình:

```
composer require laravel/tinker
```

> Tìm kiếm giao diện người dùng đồ họa để tương tác với ứng dụng Laravel của bạn? Hãy thử kiểm tra Tinkerwell!

Tinker cho phép bạn tương tác với toàn bộ ứng dụng Laravel của mình trên dòng lệnh, bao gồm các mô hình Eloquent, job, event và hơn thế nữa. Để vào môi trường Tinker, hãy chạy lệnh `tinker` Artisan:

```
php artisan tinker
```

Bạn có thể publish tập tin cấu hình của Tinker bằng lệnh `vendor:publish`:

```
php artisan vendor:publish --provider="Laravel\Tinker\TinkerServiceProvider"
```

> **Chú ý:** Hàm helper `dispatch` và phương thức `dispatch` trên lớp `Dispatchable` phụ thuộc vào việc thu gom rác để đặt job vào hàng đợi. Do đó, khi sử dụng tinker, bạn nên sử dụng `Bus::dispatch` hoặc `Queue::push` để điều phối job.

Danh sách lệnh được cho phép

Tinker sử dụng danh sách "cho phép" để xác định lệnh Artisan nào được phép chạy trong shell console của nó. Mặc định, bạn có thể chạy các lệnh đã biên dịch rõ ràng, `down`, `env`, `inspire`, `migrate`, `optimize` và `up`. Nếu bạn muốn cho phép nhiều lệnh hơn, bạn có thể thêm chúng vào mảng `commands` trong tập tin cấu hình *tinker.php* của bạn:

```
'commands' => [
  // App\Console\Commands\ExampleCommand::class,
],
```

Class không được liên kết

Thông thường, Tinker tự động đặt bí danh liên kết cho các class khi bạn tương tác với chúng trong Tinker. Tuy nhiên, bạn có thể muốn không bao giờ đặt bí danh liên kết cho một số class. Bạn có thể thực hiện điều này bằng cách liệt kê các class này vào trong mảng `dont_alias` của tập tin cấu hình *tinker.php* của bạn:

```
'dont_alias' => [
  App\Models\User::class,
],
```

Viết lệnh mới

Ngoài các lệnh được cung cấp sẵn cho Artisan, bạn có thể tạo ra các lệnh tự tạo của riêng mình. Các lệnh thường được lưu trữ trong thư mục *app/Console/Commands*; tuy nhiên, bạn có thể tự do chọn vị trí lưu trữ của mình miễn là Composer có thể tải các lệnh của bạn.

Tạo lệnh

Để tạo một lệnh mới, bạn có thể sử dụng lệnh Artisan `make:command`. Lệnh này sẽ tạo một class lệnh mới trong thư mục *app/Console/Commands*. Đừng lo lắng nếu thư mục này không tồn tại trong ứng dụng của bạn - nó sẽ tự động được tạo khi lần đầu tiên bạn chạy lệnh `make:command` Artisan:

```
php artisan make:command SendEmails
```

Sau khi tạo lệnh, bạn nên xác định các giá trị thích hợp cho các thuộc tính `signature` và `description` của class. Các thuộc tính này sẽ được sử dụng khi hiển thị lệnh của bạn trên màn hình liệt kê. Thuộc tính `signature` cũng cho phép bạn xác định các đầu vào input của lệnh. Phương thức `handle` sẽ được gọi khi lệnh của bạn được thực thi. Bạn có thể đặt logic lệnh của mình trong phương thức này.

Hãy xem một lệnh ví dụ. Xin lưu ý rằng, chúng ta có thể yêu cầu bất kỳ thư viện nào chúng ta cần thông qua phương thức `handle` của lệnh. Service container của Laravel sẽ tự động đưa vào tất cả các thư viện nào được khai kiểu trong phần tham số của phương thức này:

```
<?php
namespace App\Console\Commands;
use App\Models\User;
use App\Support\DripEmailer;
use Illuminate\Console\Command;

class SendEmails extends Command
{
  /**
   * The name and signature of the console command.
   *
   * @var string
   */
```

```
    protected $signature = 'mail:send {user}';

    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Send a marketing email to a user';

    /**
     * Create a new command instance.
     *
     * @return void
     */
    public function __construct()
    {
        parent::__construct();
    }

    /**
     * Execute the console command.
     *
     * @param  \App\Support\DripEmailer  $drip
     * @return mixed
     */
    public function handle(DripEmailer $drip)
    {
        $drip->send(User::find($this->argument('user')));
    }
}
```

Để tái sử dụng mã nhiều hơn, bạn nên giữ cho các lệnh của bạn gọn nhẹ và để chúng tạm hoãn các service của ứng dụng nhằm hoàn thành nhiệm vụ của mình. Trong ví dụ trên, hãy lưu ý rằng chúng ta chèn một class service để thực hiện việc gửi e-mail.

Hàm nặc danh và lệnh

Các lệnh dựa trên hàm nặc danh cung cấp một giải pháp thay thế cho việc tạo các lệnh console dưới dạng các class. Tương tự như cách mà các hàm nặc danh trên route làm giải pháp thay thế cho controller, hãy nghĩ về các lệnh trên hàm nặc danh như một giải pháp thay thế cho các lệnh class. Trong phương thức `commands` của tệp *app/Console/Kernel.php* của bạn, Laravel sẽ tải tập tin `routes/console.php`:

```
/**
 * Register the closure based commands for the application.
 *
 * @return void
 */
protected function commands()
{
  require base_path('routes/console.php');
}
```

Mặc dù tập tin này không tạo ra được các HTTP route, nhưng nó sẽ khai báo các entry point cho các lệnh console cho ứng dụng của bạn. Trong tập tin này, bạn có thể tạo ra tất cả các lệnh console dựa trên hàm nặc danh của mình bằng cách sử dụng phương thức `Artisan::command`. Phương thức `command` chấp nhận hai đối số: signature của lệnh và một hàm xử lý các đối số và tùy chọn của lệnh:

```
Artisan::command('mail:send {user}', function ($user) {
  $this->info("Sending email to: {$user}!");
});
```

Hàm nặc danh được liên kết với đối tượng command bên dưới, vì vậy bạn có toàn quyền truy cập vào tất cả các phương thức trợ giúp mà bạn thường truy cập trên một class lệnh.

Đưa vào các thư viện

Ngoài việc nhận các đối số và tùy chọn của lệnh của bạn, các lệnh trên hàm nặc danh cũng có thể khai kiểu các thư viện bổ sung mà bạn muốn đưa vào từ service container:

```
use App\Models\User;
use App\Support\DripEmailer;
```

```
Artisan::command('mail:send {user}', function (DripEmailer $drip, $user) {
  $drip->send(User::find($user));
});
```

Mô tả lệnh

Khi tạo một lệnh dựa trên hàm nặc danh, thì bạn có thể sử dụng phương thức `purpose` để thêm phần mô tả về lệnh. Mô tả này sẽ được hiển thị khi bạn chạy lệnh `php artisan list` hoặc các lệnh trợ giúp của `php artisan help`:

```
Artisan::command('mail:send {user}', function ($user) {
  // ...
})->purpose('Send a marketing email to a user');
```

Nhận dữ liệu input

Khi viết các lệnh console, thông thường người dùng sẽ thu thập thông tin đầu vào thông qua các đối số hoặc tùy chọn. Laravel rất thuận tiện để tạo các input đầu vào mà bạn muốn được nhập từ người dùng bằng cách sử dụng thuộc tính signature trên các lệnh của bạn. Thuộc tính signature cho phép bạn chỉ định tên, các đối số và tùy chọn cho lệnh trong một cú pháp thống nhất.

Đối số

Tất cả các đối số và tùy chọn do người dùng cung cấp được bao quanh bởi dấu ngoặc móc. Trong ví dụ sau, một lệnh sẽ chỉ định một đối số bắt buộc: `user`:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'mail:send {user}';
```

Bạn cũng có thể đặt các đối số theo dạng tùy chọn hoặc chỉ định các giá trị mặc định cho các đối số:

```
// Optional argument...
mail:send {user?}

// Optional argument with default value...
mail:send {user=foo}
```

Tuỳ chọn

Các tùy chọn, như đối số, là hình thức nhập liệu từ người dùng. Các option được bắt đầu bằng hai dấu gạch ngang (`--`) khi chúng được cung cấp qua dòng lệnh. Có hai loại option: option nhận giá trị và option không nhận giá trị. Các option không nhận giá trị đóng vai trò như một "switch" boolean. Hãy xem một ví dụ cụ thể về loại option này:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'mail:send {user} {--queue}';
```

Trong ví dụ này, option `--queue` có thể được chỉ định khi gọi lệnh Artisan. Nếu `--queue` được truyền vào lệnh, thì giá trị của option sẽ là `true`. Nếu không, giá trị sẽ là `false`:

```
php artisan mail:send 1 --queue
```

Option có giá trị cụ thể

Tiếp theo, chúng ta hãy xem xét một option có một giá trị cụ thể. Nếu người dùng phải chỉ định một giá trị cụ thể cho một option, thì bạn nên đặt tên option bằng cách thêm vào sau option một dấu `=`, ví dụ:

```
/**
 * The name and signature of the console command.
 *
 * @var string
```

```
 */
protected $signature = 'mail:send {user} {--queue=}';
```

Trong ví dụ này, người ta có thể truyền một giá trị cụ thể nào đó cho option `--queue`. Nếu option không được chỉ định khi gọi lệnh, giá trị của nó sẽ là `null`:

```
php artisan mail:send 1 --queue=default
```

Bạn có thể gán giá trị mặc định cho các option bằng cách chỉ định giá trị mặc định sau tên option. Nếu không có giá trị option nào được người dùng truyền vào, thì giá trị mặc định sẽ được sử dụng thay thế:

```
mail:send {user} {--queue=default}
```

Shortcut của option

Để gán một shortcut khi chi định một option, bạn có thể chỉ định shortcut trước tên option và sử dụng ký tự `|` dưới dạng dấu phân cách để phân biệt với tên option:

```
mail:send {user} {--Q|queue}
```

Khi gọi lệnh trên thiết bị đầu cuối của bạn, các shortcut của option phải được đặt trước bằng một dấu gạch ngang(`-`):

```
php artisan mail:send 1 -Q
```

Nhập input mảng dữ liệu

Nếu bạn muốn xác định các đối số hoặc option có thể nhận nhiều giá trị đầu vào, bạn có thể sử dụng ký tự *, nó được gọi là ký tự wildcard. Đầu tiên, chúng ta hãy xem một ví dụ có một đối số giống như mô tả trong trường hợp này:

```
mail:send {user*}
```

Khi gọi lệnh này, các đối số `user` có thể được truyền tới dòng lệnh. Ví dụ: lệnh sau sẽ đặt giá trị của `user` thành một mảng với `foo` và `bar` là các giá trị của nó:

```
php artisan mail:send foo bar
```

Ký tự * này có thể được kết hợp sau một ký hiệu *đối số không bắt buộc*, nhằm cho phép việc không có hoặc có nhiều đối số:

```
mail:send {user?*}
```

Mảng Option

Khi xác định một option nào đó yêu cầu nhiều giá trị đầu vào, mỗi giá trị option được truyền đến lệnh phải được đặt trước bằng tên option:

```
mail:send {user} {--id=*}

php artisan mail:send --id=1 --id=2
```

Mô tả dữ liệu input

Bạn có thể gán mô tả cho các đối số và option bằng cách sử dụng dấu hai chấm để tách tên đối số khỏi mô tả. Nếu bạn cần thêm một ít chỗ để khai báo lệnh của mình, thì hãy trải rộng phần khai báo này trên nhiều dòng code:

```
/**
 * The name and signature of the console command.
 *
 * @var string
 */
protected $signature = 'mail:send
                        {user : The ID of the user}
                        {--queue : Whether the job should be queued}';
```

I/O của lệnh

Truy xuất đầu vào

Trong khi lệnh của bạn đang thực thi, bạn có thể sẽ cần truy cập các giá trị cho các đối số và tùy chọn được lệnh của bạn chấp nhận. Để làm như vậy, bạn có thể sử dụng các phương thức `argument` và `option`. Nếu một đối số hoặc tùy chọn không tồn tại, `null` sẽ được trả về:

```
/**
 * Execute the console command.
 *
 * @return int
 */
public function handle()
{
  $userId = $this->argument('user');

  //
}
```

Nếu bạn cần truy xuất tất cả các đối số dưới dạng một `array`, hãy gọi phương thức `arguments`:

```
$arguments = $this->arguments();
```

Các option có thể được truy xuất dễ dàng như các đối số bằng cách sử dụng phương thức `option`. Để truy xuất tất cả các option dưới dạng một mảng, hãy gọi phương thức `options`:

```
// Retrieve a specific option...
$queueName = $this->option('queue');

// Retrieve all options as an array...
$options = $this->options();
```

Ngõ nhập đầu vào

Ngoài việc hiển thị đầu ra, bạn cũng có thể yêu cầu người dùng cung cấp dữ liệu đầu vào trong quá trình thực hiện lệnh của bạn. Phương thức `ask` sẽ nhắc người dùng với câu hỏi đã cho, chấp nhận đầu vào của họ và sau đó trả lại đầu vào của người dùng về lệnh của bạn:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
  $name = $this->ask('What is your name?');
}
```

Phương thức `secret` tương tự như `ask`, nhưng đầu vào của người dùng sẽ bị ẩn trên màn hình với họ khi họ nhập vào console. Phương thức này hữu ích khi yêu cầu thông tin nhạy cảm như mật khẩu:

```
$password = $this->secret('What is the password?');
```

Yêu cầu xác nhận

Nếu bạn cần yêu cầu người dùng xác nhận đơn giản "có hoặc không", bạn có thể sử dụng phương thức `confirm`. Mặc định, phương thức này sẽ trả về `false`. Tuy nhiên, nếu người dùng nhập `y` hoặc `yes` đáp lại lời nhắc, phương thức sẽ trả về `true`.

```
if ($this->confirm('Do you wish to continue?')) {
  //
}
```

Nếu cần, bạn có thể chỉ định rằng, lời nhắc xác nhận sẽ trả về `true` theo mặc định bằng cách truyền giá trị `true` làm đối số thứ hai cho phương thức `confirm`:

```
if ($this->confirm('Do you wish to continue?', true)) {
  //
}
```

Tự động hoàn thành

Phương thức `anticipate` có thể được sử dụng để cung cấp tính năng tự động hoàn thành cho các lựa chọn khả thi. Người dùng vẫn có thể cung cấp bất kỳ câu trả lời nào, bất kể gợi ý gì cũng sẽ tự động hoàn thành:

```
$name = $this->anticipate('What is your name?', ['Taylor', 'Dayle']);
```

Ngoài ra, bạn có thể truyền một hàm làm đối số thứ hai cho phương thức `anticipate`. Hàm này sẽ được gọi mỗi khi người dùng nhập một ký tự đầu vào. Hàm này phải được cung cấp một tham số chuỗi có chứa đầu vào khả thi của người dùng và trả về một mảng các tùy chọn để tự động hoàn thành:

```
$name = $this->anticipate('What is your address?', function ($input) {
  // Return auto-completion options...
});
```

Câu hỏi nhiều lựa chọn

Nếu bạn cần cung cấp cho người dùng một tập hợp các lựa chọn được xác định trước khi đặt câu hỏi, bạn có thể sử dụng phương thức `choice`. Bạn có thể đặt chỉ mục mặc định của giá trị mảng mà sẽ được trả về nếu không có tùy chọn nào được chọn bằng cách truyền chỉ mục làm đối số thứ ba cho phương thức:

```
$name = $this->choice(
  'What is your name?',
  ['Taylor', 'Dayle'],
  $defaultIndex
);
```

Ngoài ra, phương thức `choice` còn chấp nhận đối số thứ tư và thứ năm, để xác định số lần

thử tối đa để chọn phản hồi hợp lệ và có cho phép nhiều lựa chọn hay không:

```
$name = $this->choice(
  'What is your name?',
  ['Taylor', 'Dayle'],
  $defaultIndex,
  $maxAttempts = null,
  $allowMultipleSelections = false
);
```

Viết đầu ra

Để gửi đầu ra đến console, bạn có thể sử dụng các phương thức `line`,,, và . Mỗi phương thức này sẽ sử dụng màu ANSI thích hợp cho mục đích của chúng. Ví dụ: hãy hiển thị một số thông tin chung cho người dùng. Thông thường, phương thức sẽ hiển thị trong console dưới dạng văn bản màu xanh lục:`info` `comment` `question` `warn` `error` `info`.

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
  // ...

  $this->info('The command was successful!');
}
```

Để hiển thị thông báo lỗi, hãy sử dụng phương thức `error`. Văn bản thông báo lỗi thường được hiển thị bằng màu đỏ:

```
$this->error('Something went wrong!');
```

Bạn có thể sử dụng phương thức `line` để hiển thị văn bản thuần túy, không có màu:

```php
$this->line('Display this on the screen');
```

Bạn có thể sử dụng phương thức `newLine` để hiển thị một dòng trống:

```php
// Write a single blank line...
$this->newLine();

// Write three blank lines...
$this->newLine(3);
```

Hiển thị Tables

Phương thức table giúp bạn dễ dàng định dạng chính xác nhiều hàng/cột dữ liệu như một table thực thụ trên console. Tất cả những gì bạn cần làm là cung cấp tên cột và dữ liệu cho bảng và Laravel sẽ tự động tính toán chiều rộng và chiều cao thích hợp của bảng cho bạn:

```php
use App\Models\User;

$this->table(
  ['Name', 'Email'],
  User::all(['name', 'email'])->toArray()
);
```

Thanh tiến trình

Đối với các tác vụ đang chạy, có thể hữu ích khi hiển thị thanh tiến trình thông báo cho người dùng mức độ hoàn thành của tác vụ. Sử dụng phương `withProgressBar`, Laravel sẽ hiển thị một thanh tiến trình và tăng tiến trình của nó cho mỗi lần lặp qua một giá trị có thể lặp lại nhất định:

```php
use App\Models\User;

$users = $this->withProgressBar(User::all(), function ($user) {
  $this->performTask($user);
});
```

Đôi khi, bạn có thể cần kiểm soát thủ công nhiều hơn đối với cách tăng thanh tiến trình. Đầu tiên, xác định tổng số bước mà quy trình sẽ lặp lại. Sau đó, tăng thanh tiến trình sau khi xử lý từng mục:

```php
$users = App\Models\User::all();

$bar = $this->output->createProgressBar(count($users));

$bar->start();

foreach ($users as $user) {
  $this->performTask($user);

  $bar->advance();
}

$bar->finish();
```

> Để có thêm các tùy chọn nâng cao, hãy xem tài liệu về thành phần Thanh tiến trình của Symfony.

Đăng ký lệnh

Tất cả các lệnh console của bạn được đăng ký trong class `App\Console\Kernel` của ứng dụng của bạn, là "hạt nhân console" của ứng dụng của bạn. Trong phương thức `commands` của class này, bạn sẽ thấy một lệnh gọi đến phương thức `load` của kernel. Phương thức `load` này sẽ quét thư mục *app/Console/Commands* và tự động đăng ký từng lệnh mà nó chứa với Artisan. Bạn thậm chí có thể tự do thực hiện các lệnh gọi bổ sung tới phương thức `load` để quét các thư mục khác cho các lệnh Artisan:

```php
/**
 * Register the commands for the application.
 *
 * @return void
 */
protected function commands()
{
  $this->load(__DIR__.'/Commands');
  $this->load(__DIR__.'/../Domain/Orders/Commands');

  // ...
}
```

Nếu cần, bạn có thể đăng ký các lệnh theo cách thủ công bằng cách thêm tên class của lệnh vào thuộc tính `$commands` trong class `App\Console\Kernel` của bạn. Nếu thuộc tính này chưa được khai báo trên kernel của bạn, bạn nên khai báo nó theo cách thủ công. Khi Artisan khởi động, tất cả các lệnh được liệt kê trong thuộc tính này sẽ được tải vào bởi service container và được đăng ký với Artisan:

```php
protected $commands = [
  Commands\SendEmails::class
];
```

Thực thi các lệnh theo chương trình

Đôi khi bạn có thể muốn thực hiện một lệnh Artisan bên ngoài CLI. Ví dụ: bạn có thể muốn thực hiện lệnh Artisan từ một route hoặc controller. Bạn có thể sử dụng phương thức `call` trên facade `Artisan` để thực hiện điều này. Phương thức `call` chấp nhận tên của lệnh hoặc tên class làm đối số đầu tiên của nó và một mảng tham số lệnh làm đối số thứ hai. Exit code sẽ được trả lại:

```php
use Illuminate\Support\Facades\Artisan;

Route::post('/user/{user}/mail', function ($user) {
  $exitCode = Artisan::call('mail:send', [
    'user' => $user, '--queue' => 'default'
```

```
    ]);

    //
});
```

Ngoài ra, bạn có thể truyền toàn bộ lệnh Artisan vào phương thức `call` dưới dạng một câu chuỗi:

```
Artisan::call('mail:send 1 --queue=default');
```

Truyền giá trị mảng

Nếu lệnh của bạn xác định một tùy chọn chấp nhận một mảng, bạn có thể truyền một mảng giá trị cho tùy chọn đó:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/mail', function () {
    $exitCode = Artisan::call('mail:send', [
        '--id' => [5, 13]
    ]);
});
```

Truyền giá trị Boolean

Nếu bạn cần chỉ định giá trị của một tùy chọn không chấp nhận các giá trị chuỗi, chẳng hạn như cờ `--force` trên lệnh `migrate:refresh`, thì bạn nên truyền `true` hoặc `false` dưới dạng giá trị của tùy chọn:

```
$exitCode = Artisan::call('migrate:refresh', [
    '--force' => true,
]);
```

Các lệnh của nghệ nhân xếp hàng

Bằng cách sử dụng phương thức `queue` trên facade `Artisan`, bạn thậm chí có thể xếp hàng các lệnh của Artisan để chúng được xử lý ở chế độ nền bởi lịch trình xếp hàng của bạn . Trước khi sử dụng phương thức này, hãy đảm bảo rằng bạn đã định cấu hình hàng đợi của mình và đang chạy một chương trình theo dõi hàng đợi:

```
use Illuminate\Support\Facades\Artisan;

Route::post('/user/{user}/mail', function ($user) {
  Artisan::queue('mail:send', [
    'user' => $user, '--queue' => 'default'
  ]);

  //
});
```

Sử dụng các phương thức `onConnection` và `onQueue`, bạn có thể chỉ định kết nối hoặc hàng đợi lệnh Artisan sẽ được gửi đến:

```
Artisan::queue('mail:send', [
  'user' => 1, '--queue' => 'default'
])->onConnection('redis')->onQueue('commands');
```

Gọi lệnh từ các lệnh khác

Đôi khi bạn có thể muốn gọi các lệnh khác từ lệnh Artisan hiện có. Bạn có thể làm như vậy bằng cách sử dụng phương thức `call`. Phương thức `call` chấp nhận tên lệnh và một mảng các đối số/tùy chọn của lệnh:

```
/**
 * Execute the console command.
 *
 * @return mixed
 */
public function handle()
{
  $this->call('mail:send', [
```

```
    'user' => 1, '--queue' => 'default'
  ]);

  //
}
```

Nếu bạn muốn gọi một lệnh console khác và chặn tất cả đầu ra của nó, bạn có thể sử dụng phương thức `callSilently`. Phương thức `callSilently` có cùng hình thức với phương thức `call`:

```
$this->callSilently('mail:send', [
  'user' => 1, '--queue' => 'default'
]);
```

Xử lý tín hiệu

Component *Symfony Console*, hỗ trợ Artisan console, cho phép bạn chỉ ra các tín hiệu xử lý (nếu có) mà lệnh của bạn sẽ xử lý. Ví dụ, bạn có thể chỉ ra rằng lệnh của bạn sẽ xử lý các tín hiệu `SIGINT` và `SIGTERM`.

Để bắt đầu, bạn nên thực thi interface `Symfony\Component\Console\Command\SignalableCommandInterface` trên class lệnh Artisan của mình. Interface này yêu cầu bạn khai báo hai phương thức: `getSubscribedSignals` và `handleSignal`:

```
<?php

use Symfony\Component\Console\Command\SignalableCommandInterface;

class StartServer extends Command implements SignalableCommandInterface
{
  // ...

  /**
   * Get the list of signals handled by the command.
   *
   * @return array
```

```
     */
    public function getSubscribedSignals(): array
    {
        return [SIGINT, SIGTERM];
    }

    /**
     * Handle an incoming signal.
     *
     * @param  int  $signal
     * @return void
     */
    public function handleSignal(int $signal): void
    {
        if ($signal === SIGINT) {
            $this->stopServer();

            return;
        }
    }
}
```

Như bạn có thể thấy,phương thức `getSubscribedSignals` sẽ trả về một mảng các tín hiệu mà lệnh của bạn có thể xử lý, trong khi phương thức `handleSignal` sẽ nhận tín hiệu và có thể phản hồi tương ứng.

Tùy chỉnh Stub

Các lệnh `make` của Artisan được sử dụng để tạo nhiều class khác nhau, chẳng hạn như controller, job, migration và test. Các class này được tạo ra bằng cách sử dụng các tập tin "sơ khai - stub" được điền các giá trị dựa trên đầu vào của bạn. Tuy nhiên, bạn có thể muốn thực hiện các thay đổi nhỏ đối với tập tin do Artisan tạo. Để thực hiện điều này, bạn có thể sử dụng lệnh `stub:publish` để áp dụng các tập tin sơ khai phổ biến nhất cho ứng dụng của mình nhằm để cho bạn có thể dễ dàng tùy chỉnh chúng:

```
php artisan stub:publish
```

Các bản gốc đã publish trước đó sẽ nằm trong thư mục `stubs` trên thư mục gốc của ứng dụng của bạn.Bất kỳ thay đổi nào bạn thực hiện đối với các stub này sẽ được phản ánh khi bạn tạo các class tương ứng của chúng bằng lệnh `make` của Artisan.

Sự kiện

Artisan điều hành ba sự kiện khi chạy lệnh console, lần lượt là,

1. `Illuminate\Console\Events\ArtisanStarting`,
2. `Illuminate\Console\Events\CommandStarting`,
3. `Illuminate\Console\Events\CommandFinished`.

Sự kiện `ArtisanStarting` được gửi đi ngay lập tức khi Artisan bắt đầu chạy. Tiếp theo, sự kiện `CommandStarting` được gửi ngay lập tức trước khi lệnh chạy. Cuối cùng, sự kiện `CommandFinished` sẽ được gửi đi khi một lệnh kết thúc quá trình thực thi của nó.