Laravel Artisan Console

Artisan là tên của giao diện màn hình gõ lệnh đính kèm trong Laravel. Nó cung cấp một danh sách các câu lệnh hữu ích để sử dụng trong quá trình phát triển sản phẩm. Artisan được phát triển dựa trên component Symfony Console khá mạnh mẽ. Để xem danh sách các câu lệnh được cung cấp, bạn có thể sử dụng câu lệnh `list`:

```
php artisan list
```

Mỗi câu lệnh đều có kèm theo một màn hình "help" để hiển thị và mô tả những đối số và tuỳ chọn có thể sử dụng. Để xem màn hình help, đơn giản chỉ cần gõ tên câu lệnh kèm theo từ khoá `help`:

```
php artisan help migrate
```

Tự xây dựng Artisan Console

Tạo câu lệnh Artisan mới

Ngoài việc sử dụng các câu lệnh được cung cấp sẵn, ban cũng có thể tạo câu lệnh riêng để sử dụng cho ứng dụng của bạn. Bạn có thể lưu trữ các câu lệnh riêng đó trong thư mục `app/Console/Commands`; tuy nhiên, bạn hoàn toàn thoải mái trong việc chọn vị trí lưu đặt mã nguồn các câu lệnh với điều kiện là phải khai báo tự động khởi tạo trong cấu hình của `composer.json`.

Để tạo một câu lệnh mới, bạn có thể sử dụng câu lệnh `make:console`, nó sẽ tạo ra các khung mã nguồn cơ bản để giúp bạn bắt đầu một cách dễ dàng hơn:

```
php artisan make:console SendEmails
```

Câu lệnh trên sẽ tạo một class tại `app/Console/Commands/SendEmails.php`. Khi tạo một câu lệnh, tuỳ chọn `--command` có thể được gán giá trị trên màn hình terminal

```
php artisan make:console SendEmails --command=emails:send
```

Cấu trúc câu lệnh

Khi mà câu lệnh được tạo ra, bạn nên điền vào thông tin của hai thuộc tính `signature` và `description` trong class, vì chúng sẽ được dùng để hiển thị khi mà câu lệnh `list` được thực thi

Phương thức `handle` sẽ được gọi khi mà câu lệnh được thực thi. Bạn có thể viết logic tuỳ ý trong phương thức này

Chúng ta có thể inject bất cứ dependencies nào mà chúng ta cần vào trong hàm khởi tạo của câu lệnh.

Laravel service container sẽ tự động inject tất cả các dependencies được đánh dầu trong hàm khởi tạo.

Để mã nguồn tái sử dụng tốt hơn, khuyến khích các bạn xử lý câu lệnh một cách gọn nhẹ và chuyển giao cho application services để thực hiện công việc.

```
<?php

    namespace App\Console\Commands;

    use App\User;
    use App\DripEmailer;
    use Illuminate\Console\Command;

    class SendEmails extends Command
    {
        /**
         * The name and signature of the console command.
         *
         * @var string
         */
        protected $signature = 'email:send {user}';

        /**
         * The console command description.
         *
         * @var string
         */
        protected $description = 'Send drip e-mails to a user';

        /**
         * The drip e-mail service.
         *
         * @var DripEmailer
         */
        protected $drip;

        /**
         * Create a new command instance.
         *
         * @param  DripEmailer  $drip
```

```
         * @return void
         */
        public function __construct(DripEmailer $drip)
        {
            parent::__construct();

            $this->drip = $drip;
        }

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

Thiết lập dữ liệu vào ra trong Artisan Console

Khai báo yêu cầu dữ liệu đầu vào

Thông thường chúng ta sẽ nhận dữ liệu đầu vào từ người sử dụng thông qua các đối số và tuỳ chọn khi thực hiện viết các câu lệnh console.

Laravel làm cho việc này trở nên tiện hơn khi khai báo yêu cầu dữ liệu đầu vào sử dụng thuộc tính `signature` trong câu lệnh.

Thuộc tính `signature` cho phép bạn khai báo tên, đối số, và các tuỳ chọn cho câu lệnh dưới dạng một giá trị, một biểu thức hay cú pháp tương tự khai báo route.

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

Bạn cũng có thể đặt đối số này là tuỳ chọn và cài đặt giá trị mặc định

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

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

Tuỳ chọn, như đối số, cũng là một kiểu nhập vào từ người sử dụng nhưng chúng có tiền tố là hai dấu gạch ngang (`--`) khi được viết. Chúng ta có thể khai báo tuỳ chọn trong `signature` như sau:

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

Ở ví dụ này, tuỳ chọn `--queue`có thể được chỉ định khi thực hiện gọi câu lệnh. Nếu như `--queue`được gọi, thì giá trị của tuỳ chọn này sẽ là `true`. Ngược lại, giá trị sẽ là `false`

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

Bạn cũng có thể điều chỉnh sao cho tuỳ chọn phải được gán với một giá trị bởi người dùng thông qua việc sử dụng kí hiệu `=`để cho biết là cần yêu cầu có dữ liệu nhập vào:

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

Trong ví dụ này, người sử dụng có thể truyền vào một giá trị cho tuỳ chọn:

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

Bạn cũng có thể gán giá trị mặc định cho tuỳ chọn:

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

Để thiết lập một shortcut khi khai báo tuỳ chọn, bạn có thể thêm vào tên của shortcut ngay trước tên của tuỳ chọn và dùng dấu | để ngăn cách:

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

Nếu bạn muốn đối số hay tuỳ chọn của dữ liệu đầu vào là một mảng, sử dụng dấu `*`:

```
email:send {user*}

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

Mô tả dữ liệu đầu vào

Bạn có thể gán nội dung mô tả cho các đối số và tuỳ chọn bằng việc sử dụng dấu `:` để ngăn cách:

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

Nhận dữ liệu đầu vào

Khi câu lệnh được thực thi, rõ ràng là chúng ta cần lấy được giá trị của các đối số và tuỳ chọn được nhận vào câu lệnh. Để làm được điều này, bạn cần sử dụng tới phương thức `argument` và `option`:

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

        //
    }
```

Nếu bạn cần lấy tất cả đối số truyền vào dưới dạng một `array`, sử dụng `argument` mà không truyền vào tham số nào:

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

Tuỳ chọn có thể nhận thông qua phương thức `option`. Bạn sử dụng phương thức `option` một cách tương tự với `argument`

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

// Retrieve all options...
$options = $this->option();
```

Nếu như đối số hay tuỳ chọn không tồn tại, giá trị nhận được sẽ là `null`.

Hỏi yêu cầu nhập dữ liệu

Ngoài việc hiển thị, bạn cũng có thể yêu cầu người dùng nhập dữ liệu trong quá trình thực thi câu lệnh. Phương thức `ask` sẽ yêu cầu người dùng nhập dữ liệu với câu hỏi được đưa ra, nhận dữ liệu và truyền dữ liệu nhập từ người dùng vào trong câu lệnh:

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

Phương thức `secret` tương tự như `ask` nhưng dữ liệu mà người dùng nhập khi gõ bàn phím sẽ không được hiển thị lên. Phương thức này rất hữu ích khi yêu cầu các thông tin nhạy cảm như mật khẩu:

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

Yêu cầu xác nhận

Nếu bạn đơn giản chỉ cần một xác nhận từ người sử dụng, bạn có thể sử dụng phương thức `confirm`. Mặc định thì phương thức này trả lại giá trị `false`. Tuy nhiên, nếu người dùng nhập vào `y` thì phương thức sẽ trả về `true`.

```
if ($this->confirm('Do you wish to continue? [y|N]')) {
        //
}
```

Đưa ra sự lựa chọn

Phương thức `anticipate` có thể sử dụng để cung cấp tự động hoàn thành câu lệnh với một danh sách các sự gợi ý có thể. Người dùng vẫn có thể đưa ra câu trả lời riêng không liên quan tới các gợi ý được đưa ra:

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

Nếu như bạn cần đưa ra một danh sách các sự lựa chọn, bạn có thể dùng `choice`. Người dùng sẽ chọn đáp án bằng cách nhập vào index của câu trả lời, và giá trị này sẽ được trả lại cho bạn. Bạn có thể thiệt lập giá trị chọn lựa mặc định nếu như mà người dùng không chọn gì:

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

Hiển thị nội dung

Để hiển thị nội dung ra màn hình, sử dụng các phương thức sau `line`, `info`, `comment`, `question` và `error`. Mỗi phương thức sẽ sử dụng màu ANSI tương ứng với mục đích của nó.

Để hiển thị thông tin cho người dùng, sử dụng `info`. Về cơ bản, nó sẽ hiển thị chữ màu xanh lá cây.

```
/**
     * Execute the console command.
     *
     * @return mixed
     */
    public function handle()
    {
        $this->info('Display this on the screen');
    }
```

Để hiển thị nội dung về lỗi, sử dụng `error`. Các nội dung lỗi sẽ được hiển thị bằng màu đỏ.

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

Nếu bạn muốn hiển thị nội dung đơn thuần, sử dụng `line`. Phương thức này không sử dụng bất cứ màu đặc trưng nào:

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

Phương thức `table` sẽ giúp cho việc chỉnh hiển thị các dữ liệu kiểu dòng / cột. Chỉ cần truyền vào headers và các dòng nội dung vào trong phương thức. Chiều rộng vào chiều cao sẽ được tự động tính toán căn chỉnh dựa trên dữ liệu đầu vào:

```
$headers = ['Name', 'Email'];

$users = App\User::all(['name', 'email'])->toArray();

$this->table($headers, $users);
```

Với các tác vụ chạy lâu, thì việc sử dụng một thanh tiến trình khá là hữu ích. Sử dụng `output`, chúng ta có thể khởi tạo, chạy tiến trình và dừng thanh tiến trình. Bạn có thể khai báo số lượng steps khi bắt đầu tiến trình và thực hiện chạy:

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

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

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

  $bar->advance();
}

$bar->finish();
```

Tham khảo tài liệu về Symfony Progress Bar để cập nhật thêm các tuỳ chọn khác

https://symfony.com/doc/2.7/components/console/helpers/progressbar.html

Gọi câu lệnh trên mã nguồn

Đôi lúc bạn muốn thực thi một câu lệnh Artisan nằm ngoài CLI. Bạn có thể sử dụng `call` trong `Artisan` facade để thực hiện việc này.

Phương thức `call` nhận tên của câu lệnh vào trong đối số đầu tiên, và một mảng danh sách các tham số thực thi câu lệnh ở đối số thứ hai. Mã kết quả thực thi sẽ được trả lại:

```
Route::get('/foo', function () {
        $exitCode = Artisan::call('email:send', [
            'user' => 1, '--queue' => 'default'
        ]);

        //
    });
```

Khi sử dụng `queue` trên `Artisan` facade, thì bạn của thể thực hiện câu lệnh trên hàng đợi và chúng sẽ được thực hiện ở background bởi queue workers trong ứng dụng của bạn:

```
Route::get('/foo', function () {
        Artisan::queue('email:send', [
            'user' => 1, '--queue' => 'default'
        ]);

        //
    });
```

Nếu bạn cần chỉ rõ giá trị của tuỳ chọn không nhận kiểu string, ví dụ `--force` trong câu lệnh `migrate:refresh` bạn có thể truyền vào giá trị boolean `true` hay `false`

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

Thực thi câu lệnh từ câu lệnh khác

Đôi lúc bạn muốn thực thi gọi câu lệnh từ một câu lệnh Artisan khác. Bạn có thể sử dụng phương thức `call` với tên câu lệnh và danh sách các tham số truyền vào:

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

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

        //
    }
```

Nếu bạn muốn thực thi một câu lệnh và chặn không muốn hiển thị nội dung của nó ra ngoài, bạn có thể dùng `callSilent`. Phương thức này sử dụng tương tự `call`:

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