Course - Laravel Framework

Gửi email

Gửi email không quá phức tạp. Laravel cung cấp một API email đơn giản, dễ hiểu được hỗ trợ bởi thư viện SwiftMailer phổ biến.

Tags: Mail, laravel send mail, mail laravel

Giới thiệu

Gửi email không quá phức tạp. Laravel cung cấp một API email đơn giản, dễ hiểu được hỗ trợ bởi thư viện SwiftMailer phổ biến. Laravel và SwiftMailer cung cấp driver để gửi email qua các giao thức SMTP, Mailgun, Postmark, Amazon SES và sendmail, cho phép bạn nhanh chóng gửi mail thông qua dịch vụ trên máy cục bộ hoặc trên các công nghệ đám mây mà bạn biết.

Cấu hình

Các dịch vụ email của Laravel có thể được cấu hình thông qua tập tin cấu hình *config/mail.php* của ứng dụng của bạn. Mỗi bưu phẩm được cấu hình trong tập tin này có thể có cấu hình riêng và thậm chí có riêng hẳn "phương tiện vận chuyển - transport" duy nhất của riêng nó, cho phép ứng dụng của bạn sử dụng các dịch vụ email khác nhau để gửi một số thông báo email nhất định. Ví dụ: ứng dụng của bạn có thể sử dụng Postmark để gửi email giao dịch trong khi sử dụng Amazon SES để gửi email hàng loạt.

Trong tập tin cấu hình mail của bạn, bạn sẽ tìm thấy một mảng cấu hình mailer. Mảng này chứa một mục cấu hình mẫu cho từng driver/transport chủ lực được Laravel hỗ trợ, trong khi giá trị cấu hình mặc định xác định thư nào sẽ được sử dụng theo mặc định khi ứng dụng của bạn cần gửi email.

Các yêu cầu đối với driver/transport

Các driver email được dựa trên API như Mailgun và Postmark thường đơn giản và nhanh hơn so với gửi email qua máy chủ SMTP. Bất cứ khi nào có thể, chúng tôi khuyên bạn nên sử dụng một trong các driver này. Tất cả các driver dựa trên API đều yêu cầu thư viện Guzzle HTTP, có thể được cài đặt thông qua Composer:

```
composer require guzzlehttp/guzzle
```

Mailgun

Để sử dụng Mailgun, trước tiên hãy cài đặt thư viện Guzzle HTTP. Sau đó, đặt tùy chọn mặc định trong tập tin cấu hình *config/mail.php* của bạn thành mailgun. Tiếp theo, xác minh rằng tệp cấu hình *config/services.php* của bạn chứa các tùy chọn sau:

```
'mailgun' => [
  'domain' => env('MAILGUN_DOMAIN'),
```

```
    'secret' => env('MAILGUN_SECRET'),
],
```

Nếu bạn không sử dụng Mailgun của Hoa Kỳ, bạn có thể xác định vùng bạn muốn bằng điểm cuối của vùng trong tệp cấu hình service:

```
'mailgun' => [
    'domain' => env('MAILGUN_DOMAIN'),
    'secret' => env('MAILGUN_SECRET'),
    'endpoint' => env('MAILGUN_ENDPOINT', 'api.eu.mailgun.net'),
],
```

Postmark

Để sử dụng Postmark, hãy cài đặt phương tiện truyền tải SwiftMailer của Postmark qua Composer:

```
composer require wildbit/swiftmailer-postmark
```

Tiếp theo, cài đặt thư viện Guzzle HTTP và cấu hình tùy chọn mặc định trong tệp cấu hình *config/mail.php* của bạn thành postmark. Cuối cùng, đảm bảo tập tin cấu hình *config/services.php* của bạn chứa các tùy chọn sau:

```
'postmark' => [
    'token' => env('POSTMARK_TOKEN'),
],
```

Nếu bạn muốn xác định luồng thông báo Postmark mà sẽ được sử dụng bởi một mailer nào đó, bạn có thể thêm tùy chọn cấu hình `message_stream_id` vào mảng cấu hình của mailer. Bạn có thể tìm thấy mảng cấu hình này trong tập tin cấu hình *config/mail.php* của ứng dụng:

```
'postmark' => [
  'transport' => 'postmark',
  'message_stream_id' => env('POSTMARK_MESSAGE_STREAM_ID'),
],
```

Bằng cách này, bạn cũng có thể thiết lập nhiều mailer kiểu Postmark với các luồng thông báo khác nhau.

Amazon SES

Để sử dụng driver Amazon SES, trước tiên bạn phải cài đặt Amazon AWS SDK cho PHP. Bạn có thể cài đặt thư viện này thông qua trình quản lý gói Composer:

```
composer require aws/aws-sdk-php
```

Tiếp theo, cấu hình tùy chọn mặc định trong tập tin cấu hình *config/mail.php* của bạn thành SES và đảm bảo tập tin cấu hình *config/services.php* của bạn chứa các tùy chọn sau:

```
'ses' => [
  'key' => env('AWS_ACCESS_KEY_ID'),
  'secret' => env('AWS_SECRET_ACCESS_KEY'),
  'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
],
```

Để sử dụng thông tin đăng nhập tạm thời AWS thông qua mã session token, bạn có thể thêm khóa mã token key vào cấu hình SES của ứng dụng của mình:

```
'ses' => [
  'key' => env('AWS_ACCESS_KEY_ID'),
  'secret' => env('AWS_SECRET_ACCESS_KEY'),
  'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
  'token' => env('AWS_SESSION_TOKEN'),
],
```

Nếu bạn muốn xác định các tùy chọn bổ sung mà Laravel sẽ chuyển đến phương thức `SendRawEmail` trong *AWS SDK* khi gửi email, thì bạn có thể xác định một mảng tùy chọn

trong cấu hình SES của mình:

```
'ses' => [
  'key' => env('AWS_ACCESS_KEY_ID'),
  'secret' => env('AWS_SECRET_ACCESS_KEY'),
  'region' => env('AWS_DEFAULT_REGION', 'us-east-1'),
  'options' => [
    'ConfigurationSetName' => 'MyConfigurationSet',
    'Tags' => [
      ['Name' => 'foo', 'Value' => 'bar'],
    ],
  ],
],
```

Cấu hình dự phòng

Đôi khi, một service bên thứ ba mà bạn đã cấu hình để gửi email cho ứng dụng của mình có thể không hoạt động. Trong những trường hợp này, có thể hữu ích khi xác định một hoặc nhiều cấu hình gửi mail dự phòng và sẽ được sử dụng trong trường hợp mailer chủ lực của bạn gặp sự cố.

Để thực hiện điều này, bạn nên xác định một mailer trong tập tin cấu hình mail của ứng dụng của bạn sử dụng truyền tải chuyển đổi dự phòng. Mảng cấu hình cho mailer dự phòng của ứng dụng của bạn phải chứa một loạt các mailer tham chiếu mà mail driver sẽ chọn để gửi:

```
'mailers' => [
  'failover' => [
    'transport' => 'failover',
    'mailers' => [
      'postmark',
      'mailgun',
      'sendmail',
    ],
  ],
  // ...
],
```

Khi mailer dự phòng của bạn đã được xác định, bạn nên đặt mailer này làm mailer mặc định được ứng dụng của bạn sử dụng bằng cách lấy tên của nó làm giá trị cho khóa cấu hình mặc định trong tập tin cấu hình mail của ứng dụng của bạn:

```
'default' => env('MAIL_MAILER', 'failover'),
```

Tạo mailables

Khi xây dựng các ứng dụng Laravel, mỗi loại email được gửi bởi ứng dụng của bạn được biểu thị dưới dạng một class "mailable". Những class này được lưu trữ trong thư mục *app/Mail*. Đừng lo lắng nếu bạn không thấy thư mục này trong ứng dụng của mình, vì nó sẽ tự động được tạo cho bạn khi bạn tạo class đầu tiên của mình bằng lệnh Artisan `make:mail`:

```
php artisan make:mail OrderShipped
```

Viết mailables

Khi bạn đã tạo ra một class mailable, hãy mở nó ra để chúng ta có thể khám phá nội dung của nó. Đầu tiên, hãy lưu ý rằng tất cả cấu hình của một lớp mailable được thực hiện trong phương thức `build`. Trong phương thức này, bạn có thể gọi các phương thức khác nhau như `from`, `subject`, `view`, và `attach` để cấu hình bản trình bày và gửi email.

Bạn có thể khai báo kiểu của các thư viện vào phương thức `build` của mailable.

> Service container của Laravel sẽ tự động đưa các thư viện này vào.

Cấu hình sender

Sử dụng phương thức form

Đầu tiên, hãy tìm hiểu cấu hình sender của email. Hay nói cách khác, email sẽ được gửi "từ" ai. Có hai cách để cấu hình sender. Đầu tiên, bạn có thể sử dụng phương thức `from` trong phương thức `build` của mailable class của mình:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
  return $this->from('example@example.com', 'Example')
    ->view('emails.orders.shipped');
}
```

Sử dụng địa chỉ `form` tổng bộ

Nếu ứng dụng của bạn sử dụng cùng một địa chỉ "from" cho tất cả các email của nó, thì việc gọi phương thức `from` trong mỗi mailable class mà bạn tạo có thể trở nên phức tạp. Thay vào đó, bạn có thể chỉ định địa chỉ chung "form" trong tập tin cấu hình *config/mail.php* của mình. Địa chỉ này sẽ được sử dụng nếu không có địa chỉ "from" nào khác được chỉ định trong mailable class:

```
'from' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

Ngoài ra, bạn cũng có thể xác định địa chỉ "reply_to" tổng bộ trong tập tin cấu hình *config/mail.php* của mình:

```
'reply_to' => ['address' => 'example@example.com', 'name' => 'App Name'],
```

Cấu hình hiển thị nội dụng mail

Trong phương thức `build` của mailable class, bạn có thể sử dụng phương thức `view` để chỉ định mail template nào nên được sử dụng khi hiển thị nội dung của email. Vì mỗi email thường sử dụng Blade template để hiển thị nội dung của nó, bạn có đầy đủ quyền và sự thuận lợi từ công cụ tạo Blade template khi xây dựng HTML cho email của mình:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
  return $this->view('emails.orders.shipped');
}
```

> Bạn có thể muốn tạo một thư mục `resources/views/email` để chứa tất cả các mail template của bạn; tuy nhiên, bạn có thể tự do đặt chúng ở bất cứ đâu bạn muốn trong thư mục `resources/views` của mình.

Các email kiểu plain text

Nếu bạn muốn xác định phiên bản plain text cho email, bạn có thể sử dụng phương thức `text`. Giống như phương thức `view`, phương thức `text` chấp nhận một tên mẫu sẽ được sử dụng để hiển thị nội dung của email. Bạn có thể tự do xác định cả phiên bản HTML và plain text cho email của mình:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
  return $this->view('emails.orders.shipped')
    ->text('emails.orders.shipped_plain');
```

```
}
```

Dữ liệu hiển thị

Thông qua các thuộc tính public

Thông thường, bạn sẽ muốn truyền một số dữ liệu vào nội dung hiển thị của mình mà bạn có thể sử dụng khi hiển thị HTML của email. Có hai cách bạn có thể cung cấp dữ liệu cho nội dung hiển thị của mình. Đầu tiên, bất kỳ thuộc tính public nào được xác định trên mailable class của bạn sẽ tự động được cung cấp cho nội dung hiển thị. Vì vậy, ví dụ: bạn có thể truyền dữ liệu vào constructor của mailable class và đặt dữ liệu đó vào các thuộc tính public được khai báo trên class này:

```php
<?php

namespace App\Mail;

use App\Models\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
  use Queueable, SerializesModels;

  /**
   * The order instance.
   *
   * @var \App\Models\Order
   */
  public $order;

  /**
   * Create a new message instance.
   *
   * @param  \App\Models\Order  $order
```

```
     * @return void
     */
    public function __construct(Order $order)
    {
        $this->order = $order;
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('emails.orders.shipped');
    }
}
```

Khi dữ liệu đã được đặt vào trong thuộc tính public, nó sẽ tự động có sẵn trong nội dung hiển thị của bạn, vì vậy bạn có thể truy cập nó giống như truy cập vào bất kỳ dữ liệu nào khác trong các Blade template của bạn:

```
<div>
    Price: {{ $order->price }}
</div>
```

Sử dụng phương thức `with`

Nếu bạn muốn tùy chỉnh định dạng của dữ liệu email của mình trước khi nó được gửi đến template, bạn có thể truyền dữ liệu của mình vào phần hiển thị theo cách thủ công thông qua phương thức `with`. Thông thường, bạn vẫn sẽ truyền dữ liệu qua constructor của mailable class; tuy nhiên, bạn nên đặt dữ liệu này vào thuộc tính protected hoặc thuộc tính private để dữ liệu không cung cấp tự động cho mail template. Sau đó, khi gọi phương thức `with`, ta hãy truyền một mảng dữ liệu mà bạn muốn cung cấp cho mail template:

```
<?php
```

```php
namespace App\Mail;

use App\Models\Order;
use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable
{
  use Queueable, SerializesModels;

  /**
   * The order instance.
   *
   * @var \App\Models\Order
   */
  protected $order;

  /**
   * Create a new message instance.
   *
   * @param  \App\Models\Order  $order
   * @return void
   */
  public function __construct(Order $order)
  {
    $this->order = $order;
  }

  /**
   * Build the message.
   *
   * @return $this
   */
  public function build()
  {
    return $this->view('emails.orders.shipped')
      ->with([
```

```
            'orderName' => $this->order->name,
            'orderPrice' => $this->order->price,
        ]);
    }
}
```

Khi dữ liệu đã được truyền đến phương thức `with`, thì nó sẽ tự động có sẵn trong phần hiển thị của bạn, vì vậy bạn có thể truy cập nó giống như truy cập vào bất kỳ dữ liệu nào khác trong các mẫu Blade template của bạn:

```
<div>
    Price: {{ $orderPrice }}
</div>
```

Attachment (Đính kèm tập tin)

Để thêm tệp đính kèm vào email, hãy sử dụng phương thức attach trong phương thức build của mailable class. Phương thức attach chấp nhận đường dẫn đầy đủ đến tập tin làm đối số đầu tiên của nó:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
        ->attach('/path/to/file');
}
```

Khi đính kèm tệp vào thư, bạn cũng có thể chỉ định tên hiển thị và / hoặc kiểu MIME bằng cách truyền một mảng làm đối số thứ hai cho phương thức `attach`:

```
/**
 * Build the message.
```

```
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('emails.orders.shipped')
            ->attach('/path/to/file', [
                'as' => 'name.pdf',
                'mime' => 'application/pdf',
            ]);
    }
```

Kèm tập tin từ đĩa cục bộ

Nếu bạn đã lưu trữ một tập tin trên một trong các đĩa hệ thống tập tin của mình, bạn có thể đính kèm tập tin đó vào email bằng phương pháp `attachFromStorage`:

```
    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('emails.orders.shipped')
            ->attachFromStorage('/path/to/file');
    }
```

Nếu cần, bạn có thể chỉ định tên tập tin đính kèm và các tùy chọn bổ sung bằng cách sử dụng các đối số thứ hai và thứ ba cho phương thức `attachFromStorage`:

```
    /**
     * Build the message.
     *
     * @return $this
     */
```

```
public function build()
{
  return $this->view('emails.orders.shipped')
    ->attachFromStorage('/path/to/file', 'name.pdf', [
      'mime' => 'application/pdf'
    ]);
}
```

Phương thức `attachFromStorageDisk` có thể được sử dụng nếu bạn cần chỉ định một đĩa lưu trữ khác với đĩa mặc định của mình:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
  return $this->view('emails.orders.shipped')
    ->attachFromStorageDisk('s3', '/path/to/file');
}
```

Đính kèm dữ liệu thô

Phương thức `attachData` có thể được sử dụng để đính kèm một chuỗi byte thô dưới dạng tập tin đính kèm. Ví dụ: bạn có thể sử dụng phương pháp này nếu bạn đã tạo một tập tin PDF trong bộ nhớ và muốn đính kèm nó vào email mà không cần ghi nó vào đĩa. Phương thức `attachData` chấp nhận các byte dữ liệu thô làm đối số đầu tiên của nó, tên của tập tin làm đối số thứ hai và một mảng tùy chọn làm đối số thứ ba của nó:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
```

```
{
  return $this->view('emails.orders.shipped')
    ->attachData($this->pdf, 'name.pdf', [
      'mime' => 'application/pdf',
    ]);
}
```

Đính kèm trực tiếp

Nhúng hình ảnh trực tiếp vào email của bạn thường rất cồng kềnh; tuy nhiên, Laravel cung cấp một cách thuận tiện để đính kèm hình ảnh vào email của bạn. Để nhúng hình ảnh trực tiếp, hãy sử dụng phương thức `embed` trên biến `$message` trong mẫu email của bạn. Laravel sẽ tự động cung cấp biến `$message` cho tất cả các mẫu email template của bạn, vì vậy bạn không cần phải lo lắng về việc truyền nó theo cách thủ công:

```
<body>
  Here is an image:

  <img src="{{ $message->embed($pathToImage) }}">
</body>
```

> **Chú ý:** Biến `$message` không có sẵn trong các mẫu tin nhắn với plain text vì các tin nhắn kiểu này không sử dụng tập tin đính kèm trực tiếp.

Nhúng dữ liệu thô

Nếu bạn đã có một chuỗi dữ liệu hình ảnh thô mà bạn muốn nhúng vào mẫu email, bạn có thể gọi phương thức `embedData` trên biến `$message`. Khi gọi phương thức `embedData`, bạn sẽ cần phải cung cấp tên tập tin mà sẽ được gán cho hình ảnh được nhúng:

```
<body>
  Here is an image from raw data:

  <img src="{{ $message->embedData($data, 'example-image.jpg') }}">
</body>
```

Các đối tượng có thể đính kèm

Mặc dù đính kèm tập tin vào thư thông qua các chuỗi đường dẫn đơn giản thường là đủ, nhưng trong nhiều trường hợp, các thực thể có thể đính kèm trong ứng dụng của bạn lại được thể hiện bằng các class. Ví dụ: nếu ứng dụng của bạn đang đính kèm hình ảnh vào tin nhắn, thì ứng dụng của bạn cũng có thể có một model `Photo` tượng trưng cho tấm ảnh đó. Trong trường hợp đó, có phải là không thuận tiện khi chỉ truyền model `Photo` vào phương thức `attach` không? Vì vậy, các đối tượng có thể đính kèm sẽ cho phép bạn làm điều đó.

Để bắt đầu, hãy triển khai interface `Illuminate\Contracts\Mail\Attachable` trên đối tượng sẽ có thể truy cập trong email. Interface này chỉ ra rằng class của bạn sẽ phải khai báo một phương thức `toMailAttachment` và nó trả về một đối tượng `Illuminate\Mail\Attachment`:

```php
<?php

namespace App\Models;

use Illuminate\Contracts\Mail\Attachable;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Mail\Attachment;

class Photo extends Model implements Attachable
{
  /**
   * Get the attachable representation of the model.
   *
   * @return \Illuminate\Mail\Attachment
   */
  public function toMailAttachment()
  {
    return Attachment::fromPath('/path/to/file');
  }
}
```

Khi bạn đã xác định được đối tượng có thể đính kèm của mình, bạn có thể chỉ cần truyền đối tượng đó vào phương thức `attach` khi tạo một thông báo email:

```php
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
  return $this->view('photos.resized')
    ->attach($this->photo);
}
```

Dữ liệu đính kèm có thể được lưu trữ trên dịch vụ lưu trữ từ xa như Amazon S3. Vì vậy, Laravel cũng cho phép bạn tạo các phiên bản đính kèm từ dữ liệu được lưu trữ trên một trong các đĩa hệ thống file của ứng dụng của bạn:

```php
// Create an attachment from a file on your default disk...
return Attachment::fromStorage($this->path);

// Create an attachment from a file on a specific disk...
return Attachment::fromStorageDisk('backblaze', $this->path);
```

Ngoài ra, bạn có thể tạo các phiên bản đính kèm thông qua dữ liệu mà bạn có trong bộ nhớ. Để thực hiện điều này, hãy truyền một hàm nặc danh vào phương thức `fromData`. Hàm này sẽ trả về dữ liệu thô nói về phần đính kèm:

```php
return Attachment::fromData(fn () => $this->content, 'Photo Name');
```

Laravel cũng cung cấp các phương pháp bổ sung mà bạn có thể sử dụng để điều chỉnh các tập tin đính kèm của mình. Ví dụ: bạn có thể sử dụng các phương thức `as` và `withMime` để tùy biến tên tập tin và kiểu MIME:

```php
return Attachment::fromPath('/path/to/file')
  ->as('Photo Name')
  ->withMime('image/jpeg');
```

Tag và Metadata

Một số nhà cung cấp email bên thứ ba như Mailgun và Postmark hỗ trợ "tag" và "metadata" trong mail, những thứ này được sử dụng để nhóm và theo dõi các email do ứng dụng của bạn gửi. Bạn có thể thêm tag và metadata vào email thông qua các phương thức `tag` và `metadata`:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
  return $this->view('emails.orders.shipped')
    ->tag('shipment')
    ->metadata('order_id', $this->order->id);
}
```

Nếu ứng dụng của bạn đang sử dụng driver Mailgun, bạn có thể tham khảo tài liệu của Mailgun để biết thêm thông tin về tag và metadata. Tương tự như vậy, tài liệu Postmark cũng có thể được tham khảo để biết thêm thông tin về việc hỗ trợ tag và metadata của chúng.

Nếu ứng dụng của bạn đang sử dụng Amazon SES để gửi email, bạn nên sử dụng phương thức `metadata` để gắn các "tag" SES vào email.

Điều chỉnh Symfony Message

Phương thức `withSymfonyMessage` của mailable class đã sẵn có cho phép bạn đăng ký một hàm nặc danh sẽ được gọi với đối tượng Symbol Message trước khi gửi thông báo. Điều này mang lại cho bạn cơ hội điều chỉnh một cách chuyên sâu message trước khi nó được gửi đi:

```
use Symfony\Component\Mime\Email;

/**
 * Build the message.
```

```
     *
     * @return $this
     */
    public function build()
    {
        $this->view('emails.orders.shipped');

        $this->withSymfonyMessage(function (Email $message) {
            $message->getHeaders()->addTextHeader(
                'Custom-Header', 'Header Value'
            );
        });

        return $this;
    }
```

Markdown Mailables

Markdown mailable cho phép bạn tận dụng các template và component được dựng sẵn của mail notification trong mailable của bạn. Vì các tin nhắn được viết bằng Markdown, nên Laravel có thể hiển thị các mẫu HTML đẹp và responsive cho các tin nhắn trong khi cũng tự động tạo một bản đối chiếu với văn bản thuần túy.

Tạo ra markdown mailable

Để tạo một mailable với một template Markdown tương ứng, bạn có thể sử dụng thêm tùy chọn `--markdown` của lệnh Artisan `make:mail`:

```
php artisan make:mail OrderShipped --markdown=emails.orders.shipped
```

Sau đó, khi đã cấu hình mailable trong phương thức build của nó, thì hãy gọi phương thức `markdown` thay vì phương thức `view`. Phương thức `markdown` chấp nhận tên của template Markdown và một mảng dữ liệu tùy chọn để cung cấp dữ liệu cho template:

```
    /**
     * Build the message.
     *
```

```
     * @return $this
     */
    public function build()
    {
        return $this->from('example@example.com')
                    ->markdown('emails.orders.shipped', [
                        'url' => $this->orderUrl,
                    ]);
    }
```

Viết nội dung markdown

Các class mailables kiểu Markdown sử dụng kết hợp các component Blade và cú pháp Markdown cho phép bạn dễ dàng tạo các thông điệp mail trong khi tận dụng các component UI email được dựng sẵn trong Laravel:

```
@component('mail::message')
# Order Shipped

Your order has been shipped!

@component('mail::button', ['url' => $url])
View Order
@endcomponent

Thanks,<br>
{{ config('app.name') }}
@endcomponent
```

> Không sử dụng thụt lề quá mức khi viết email với Markdown. Theo tiêu chuẩn Markdown, trình phân tích cú pháp Markdown sẽ hiển thị nội dung được thụt lề dưới dạng các khối mã.

Các component kiểu Button

Component kiểu button sẽ hiển thị một nút liên kết được căn giữa. Component chấp nhận

hai đối số, một là url và một là màu do bạn chọn. Màu được hỗ trợ là primary, success và error. Bạn có thể thêm nhiều component button vào một tin nhắn theo ý bạn muốn:

```
@component('mail::button', ['url' => $url, 'color' => 'success'])
View Order
@endcomponent
```

Các component kiểu Panel

Component kiểu Panel sẽ hiển thị khối văn bản nào đó trong một bảng có màu nền khác với phần thông điệp của nội dung mail. Điều này cho phép bạn thu hút sự chú ý của người đọc vào một khối văn bản cần nhấn mạnh:

```
@component('mail::panel')
This is the panel content.
@endcomponent
```

Các component kiểu Table

Component kiểu Table cho phép bạn chuyển đổi bảng Markdown thành một bảng HTML. Component chấp nhận bảng Markdown làm nội dung của nó. Căn chỉnh cột bảng bằng cú pháp Markdown mặc định như sau:

```
@component('mail::table')
| Laravel       | Table         | Example  |
| ------------- |:-------------:| --------:|
| Col 2 is      | Centered      | $10      |
| Col 3 is      | Right-Aligned | $20      |
@endcomponent
```

Điều chỉnh các component

Bạn có thể xuất tất cả các component Markdown trong mail sang ứng dụng của riêng mình để tùy chỉnh. Để xuất các component này, hãy sử dụng lệnh Artisan `vendor:publish` để xuất bản tag tài nguyên `laravel-mail`:

```
php artisan vendor:publish --tag=laravel-mail
```

Lệnh này sẽ xuất bản các mail component Markdown vào thư mục *resources/views/vendor/mail*. Thư mục `mail` sẽ chứa một thư mục `html` và một thư mục `text`, mỗi thư mục chứa các đại diện tương ứng với các component. Bạn có thể tự do tùy chỉnh các component này theo cách bạn muốn.

Tùy chỉnh css của mail

Sau khi xuất các component, thì thư mục `resources/views/vendor/mail/html/themes` sẽ chứa tập tin `default.css`. Bạn có thể tùy chỉnh CSS trong tập tin này và các style của css sẽ tự động được chuyển đổi thành các kiểu CSS trực tiếp trong các HTML của Markdown của bạn trong mail.

Nếu bạn muốn xây dựng một theme hoàn toàn mới cho các component Markdown của Laravel, bạn có thể đặt một tập tin CSS trong thư mục *html/themes*. Sau khi đặt tên và lưu tập tin CSS của bạn, hãy cập nhật tùy chọn theme trong tập tin cấu hình *config/mail.php* của ứng dụng để khớp với tên theme mới tạo của bạn.

Để tùy chỉnh theme cho một email sao cho nó có thể gửi riêng lẻ, thì bạn có thể đặt thuộc tính `$theme` của mailable class với tên của theme mà sẽ được sử dụng khi gửi mail đó.

Gửi mail

Để gửi tin nhắn, hãy sử dụng phương thức `to` trên facade `Mail`. Phương thức `to` chấp nhận một địa chỉ email, một cá thể người dùng hoặc một tập hợp người dùng. Nếu bạn truyền một đối tượng hoặc một tập hợp các đối tượng, thì mailer sẽ tự động sử dụng thuộc tính `email` và `name` của chúng khi xác định người nhận email, vì vậy hãy đảm bảo các thuộc tính này được cài trên các đối tượng của bạn. Khi bạn đã chỉ định người nhận mail, thì bạn có thể truyền một đối tượng của mailable class vào phương thức `send`:

```
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Mail\OrderShipped;
use App\Models\Order;
use Illuminate\Http\Request;
```

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

class OrderShipmentController extends Controller
{
  /**
   * Ship the given order.
   *
   * @param  \Illuminate\Http\Request  $request
   * @return \Illuminate\Http\Response
   */
  public function store(Request $request)
  {
    $order = Order::findOrFail($request->order_id);

    // Ship the order...

    Mail::to($request->user())->send(new OrderShipped($order));
  }
}
```

Bạn không bị giới hạn trong việc xác định người nhận "to" khi gửi tin nhắn. Bạn có thể tự do đặt người nhận với "to", "cc" và "bcc" bằng cách xâu chuỗi các phương thức tương ứng của chúng lại với nhau:

```php
Mail::to($request->user())
  ->cc($moreUsers)
  ->bcc($evenMoreUsers)
  ->send(new OrderShipped($order));
```

Lặp qua từng người nhận

Đôi khi, bạn có thể cần gửi mail đến danh sách người nhận bằng cách lặp lại một loạt các địa chỉ email/người nhận. Tuy nhiên, vì phương thức `to` thêm các địa chỉ email vào danh sách người nhận của mailable, mỗi lần lặp qua vòng lặp sẽ gửi một email khác đến mọi người nhận trước đó. Do đó, bạn phải luôn tạo lại đối tượng mailable cho từng người nhận:

```php
foreach (['taylor@example.com', 'dries@example.com'] as $recipient) {
  Mail::to($recipient)->send(new OrderShipped($order));
}
```

Gửi mail với một mailer cụ thể

Theo mặc định, Laravel sẽ gửi email bằng cách sử dụng mailer được cấu hình như mailer mặc định trong tập tin cấu hình mail của ứng dụng của bạn. Tuy nhiên, bạn có thể sử dụng phương thức `mailer` để gửi thư bằng cách sử dụng cấu hình mailer cụ thể:

```php
Mail::mailer('postmark')
  ->to($request->user())
  ->send(new OrderShipped($order));
```

Xếp hàng chờ cho Mail

Lên hàng chờ cho mail

Vì việc gửi email có thể ảnh hưởng đến hiệu xuất hoạt động của ứng dụng của bạn, nhiều nhà phát triển chọn cách xếp hàng email để gửi trong nền. Laravel làm cho việc này trở nên dễ dàng bằng cách sử dụng API hàng đợi hợp nhất được tích hợp sẵn của nó. Để lên hàng chờ cho việc gửi mail, hãy sử dụng phương thức `queue` trong facade Mail sau khi chỉ định người nhận mail:

```php
Mail::to($request->user())
  ->cc($moreUsers)
  ->bcc($evenMoreUsers)
  ->queue(new OrderShipped($order));
```

Phương thức này sẽ tự động thực hiện việc đẩy một tác vụ vào hàng chờ nhằm để cho message được gửi ở chế độ ngầm. Bạn sẽ cần phải cấu hình hàng chờ của mình trước khi sử dụng tính năng này.

Trì hoãn hàng chờ

Nếu bạn muốn trì hoãn việc gửi một email đã được xếp hàng đợi, bạn có thể sử dụng phương thức `later`. Đối với đối số đầu tiên của phương thức later, thì phương thức `later` chấp nhận một cá thể `DateTime` cho biết khi nào thông báo sẽ được gửi:

```
Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->later(now()->addMinutes(10), new OrderShipped($order));
```

Thêm vào hàng đợi cụ thể

Vì tất cả các mailable class được tạo bằng lệnh `make:mail` có sử dụng interface `Illuminate\Bus\Queuable`, nên bạn có thể gọi các phương thức `onQueue` và `onConnection` trên bất kỳ cá thể mailable class nào, điều này cho phép bạn chỉ định kết nối và tên hàng đợi cho message:

```
$message = (new OrderShipped($order))
  ->onConnection('sqs')
  ->onQueue('emails');

Mail::to($request->user())
  ->cc($moreUsers)
  ->bcc($evenMoreUsers)
  ->queue($message);
```

Hàng chờ mặc định

Nếu bạn có các mailable mà bạn muốn nó luôn được xếp vào hàng chờ, bạn có thể triển khai hợp đồng `ShouldQueue` trên lớp đó. Bây giờ, ngay cả khi bạn gọi phương thức `send` khi gửi thư, mailable sẽ vẫn được xếp hàng đợi vì nó sẽ thực thi contract:

```
use Illuminate\Contracts\Queue\ShouldQueue;

class OrderShipped extends Mailable implements ShouldQueue
{
  //
```

```
}
```

Transactions cơ sở dữ liệu & mailable được xếp hàng đợi

Khi các mailables xếp hàng đợi được gửi đi trong các transaction của cơ sở dữ liệu, chúng có thể được hàng đợi xử lý trước khi transaction của cơ sở dữ liệu được thông qua. Khi điều này xảy ra, bất kỳ cập nhật nào mà bạn đã thực hiện cho các model hoặc record của cơ sở dữ liệu trong suốt quá trình transaction của cơ sở dữ liệu rất có thể chưa được cập nhật thực tế trong cơ sở dữ liệu. Ngoài ra, bất kỳ model hoặc record cơ sở dữ liệu nào được tạo trong giao dịch có thể không tồn tại trong cơ sở dữ liệu. Nếu có thể gửi thư của bạn phụ thuộc vào các mô hình này, lỗi không mong muốn có thể xảy ra khi công việc gửi thư có thể xếp hàng đợi được xử lý.

Nếu tùy chọn cấu hình `after_commit` của kết nối hàng đợi của bạn được đặt thành `false`, bạn vẫn có thể cho một mailable được xếp hàng đợi cụ thể nào đó được gửi đi, sau khi tất cả các transaction của cơ sở dữ liệu mở đã được cam kết bằng cách gọi phương thức `afterCommit` khi gửi mail message:

```
Mail::to($request->user())->send(
  (new OrderShipped($order))->afterCommit()
);
```

Ngoài ra, bạn có thể gọi phương thức `afterCommit` từ constructor của mailable của bạn:

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;

class OrderShipped extends Mailable implements ShouldQueue
{
  use Queueable, SerializesModels;
```

```
    /**
     * Create a new message instance.
     *
     * @return void
     */
    public function __construct()
    {
        $this->afterCommit();
    }
}
```

Để tìm hiểu thêm về cách giải quyết những vấn đề này, vui lòng xem lại tài liệu liên quan đến các công việc được xếp hàng đợi và giao dịch cơ sở dữ liệu.

Hiển thị Mailable

Đôi khi bạn có thể muốn nhìn vào nội dung HTML của một mailable mà không cần phải gửi nó đi đâu hết. Để thực hiện điều này, bạn có thể gọi phương thức `render` của mailable. Phương thức này sẽ trả về nội dung được định dạng HTML của mailable gửi dưới dạng một chuỗi:

```
use App\Mail\InvoicePaid;
use App\Models\Invoice;

$invoice = Invoice::find(1);

return (new InvoicePaid($invoice))->render();
```

Hiển thị Mailable trên trình duyệt

Khi thiết kế mẫu template cho mail, thật tiện lợi khi có thể nhanh chóng xem trước mail template đó trong trình duyệt, giống như các Blade template điển hình khác. Vì lý do này, Laravel cho phép bạn trả lại bất kỳ mailable nào trực tiếp từ driver hoặc route. Khi một mailable được trả lại, nó sẽ được hiển thị và hiển thị trong trình duyệt, cho phép bạn nhanh chóng xem trước thiết kế của nó mà không cần gửi nó đến một địa chỉ email thực tế:

```php
Route::get('/mailable', function () {
  $invoice = App\Models\Invoice::find(1);

  return new App\Mail\InvoicePaid($invoice);
});
```

Chú ý: Các tệp đính kèm nội tuyến sẽ không được hiển thị khi xem trước một tập tin mailable trong trình duyệt của bạn. Để xem trước các mailables này, bạn nên gửi chúng đến một ứng dụng kiểm tra email như MailHog hoặc HELO.

Bản địa hóa mail

Laravel cho phép bạn gửi mailables ở một ngôn ngữ khác với ngôn ngữ hiện tại của request và thậm chí sẽ ghi nhớ ngôn ngữ này nếu các mail được xếp trong hàng chờ.

Để thực hiện điều này, facade Mail cung cấp phương thức locale để đặt ngôn ngữ mong muốn. Ứng dụng sẽ thay đổi thành ngôn ngữ cần chuyển khi template của mailable đang được đánh giá và sau đó chuyển ngược về ngôn ngữ trước đó khi quá trình đánh giá hoàn tất:

```php
Mail::to($request->user())->locale('es')->send(
  new OrderShipped($order)
);
```

Ngôn ngữ ưa thích của người dùng

Đôi khi, các ứng dụng muốn lưu trữ ngôn ngữ ưa thích của mỗi người dùng. Bằng cách triển khai constract `HasLocalePreference` trên một hoặc nhiều model của bạn, bạn có thể cho ứng dụng Laravel sử dụng ngôn ngữ đã được lưu trữ khi gửi mail đi:

```php
use Illuminate\Contracts\Translation\HasLocalePreference;

class User extends Model implements HasLocalePreference
{
  /**
   * Get the user's preferred locale.
```

```
     *
     * @return string
     */
    public function preferredLocale()
    {
        return $this->locale;
    }
}
```

Khi bạn đã triển khai interface này, thì Laravel sẽ tự động sử dụng ngôn ngữ ưa thích đã được lưu khi gửi mail và thông báo tới model. Do đó, không cần gọi phương thức `locale` khi sử dụng interface này:

```
Mail::to($request->user())->send(new OrderShipped($order));
```

Kiểm tra mail

Laravel cung cấp một số phương thức dùng để kiểm tra xem mail của bạn có chứa nội dung mà bạn mong đợi hay không. Các phương thức này là: assertSeeInHtml, assertDontSeeInHtml, assertSeeInOrderInHtml, assertSeeInText, assertDontSeeInText và assertSeeInOrderInText.

Như bạn có thể mong đợi, các xác nhận "HTML" xác nhận rằng phiên bản HTML của mailable của bạn có chứa một chuỗi nhất định, trong khi các xác nhận "text" khẳng định rằng phiên bản văn bản thuần túy của mailable của bạn chứa một chuỗi nhất định:

```
use App\Mail\InvoicePaid;
use App\Models\User;

public function test_mailable_content()
{
    $user = User::factory()->create();

    $mailable = new InvoicePaid($user);

    $mailable->assertSeeInHtml($user->email);
    $mailable->assertSeeInHtml('Invoice Paid');
```

```
    $mailable->assertSeeInOrderInHtml(['Invoice Paid', 'Thanks']);

    $mailable->assertSeeInText($user->email);
    $mailable->assertSeeInOrderInText(['Invoice Paid', 'Thanks']);
}
```

Kiểm tra tính năng gửi mail

Chúng tôi khuyên bạn nên kiểm tra nội dung của mailable một cách tách biệt với các bài kiểm tra của bạn để xác minh rằng một mailable cụ thể nào đó đã được "gửi" đến một người dùng cụ thể nào đó hay chưa. Để tìm hiểu cách kiểm tra xem thư có được gửi đi hay không, thi hãy xem tài liệu của chúng tôi về Mail fake.

Mail và Môi trường local

Khi phát triển một ứng dụng gửi email, có thể bạn thực sự không muốn gửi email đến các địa chỉ email trực tiếp. Laravel cung cấp một số cách để "vô hiệu hóa" việc gửi email thực tế trong quá trình phát triển cục bộ.

Log Driver

Thay vì gửi email của bạn, driver log mail sẽ ghi tất cả các thông báo email vào tập tin nhật ký của bạn để kiểm tra. Thông thường, driver này sẽ chỉ được sử dụng trong quá trình phát triển ở cục bộ. Để biết thêm thông tin về cách cấu hình ứng dụng của bạn cho mỗi môi trường, hãy xem tài liệu về cấu hình.

HELO/Mailtrap/MailHog

Ngoài ra, bạn có thể sử dụng một dịch vụ như HELO hoặc Mailtrap và driver smtp để gửi email của bạn đến một hộp thư "dummy" nơi bạn có thể giả định chúng như một ứng dụng email thực. Cách tiếp cận này có lợi ích là cho phép bạn thực sự kiểm tra các email cuối cùng trong chương trình xem thư của Mailtrap.

Nếu bạn đang sử dụng Laravel Sail, bạn có thể xem trước thư của mình bằng MailHog. Khi Sail đang chạy, bạn có thể truy cập giao diện MailHog tại: `http://localhost:8025`.

Sử dụng địa chỉ chung

Cuối cùng, bạn có thể chỉ định một địa chỉ chung bằng cách gọi phương thức `alwaysTo`

trong facade `Mail`. Thông thường, phương thức này sẽ được gọi từ phương thức boot của một trong các service provider của bạn:

```
use Illuminate\Support\Facades\Mail;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
  if ($this->app->environment('local')) {
    Mail::alwaysTo('taylor@example.com');
  }
}
```

Event

Laravel kích hoạt hai event trong quá trình gửi mail. `MessageSending` được kích hoạt trước khi một tin nhắn được gửi đi, trong khi `MessageSent` được kích hoạt sau khi một tin nhắn đã được gửi đi. Hãy nhớ rằng, những event này được kích hoạt khi mail đang được gửi đi, không phải khi nó được xếp hàng đợi. Bạn có thể đăng ký event listener cho event này trong nhà cung cấp dịch vụ `App\Providers\EventServiceProvider` của mình:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
  'Illuminate\Mail\Events\MessageSending' => [
    'App\Listeners\LogSendingMessage',
  ],
  'Illuminate\Mail\Events\MessageSent' => [
    'App\Listeners\LogSentMessage',
  ],
```

```
];
```

Tùy biến cơ chế truyền tải mail

Laravel bao gồm nhiều cơ chế truyền tải mail; tuy nhiên, bạn có thể muốn viết các cơ chế vận chuyển mail của riêng mình để gửi email với các dịch vụ khác mà Laravel không hỗ trợ. Để bắt đầu, hãy tạo một class mở rộng từ class `Symfony\Component\Mailer\Transport\AbstractTransport`. Sau đó, triển khai các phương thức `doSend` và `__toString()` trên phương tiện của bạn:

```
use MailchimpTransactional\ApiClient;
use Symfony\Component\Mailer\SentMessage;
use Symfony\Component\Mailer\Transport\AbstractTransport;
use Symfony\Component\Mime\MessageConverter;

class MailchimpTransport extends AbstractTransport
{
  /**
   * The Mailchimp API client.
   *
   * @var \MailchimpTransactional\ApiClient
   */
  protected $client;

  /**
   * Create a new Mailchimp transport instance.
   *
   * @param  \MailchimpTransactional\ApiClient  $client
   * @return void
   */
  public function __construct(ApiClient $client)
  {
    $this->client = $client;
  }

  /**
   * {@inheritDoc}
```

```
     */
    protected function doSend(SentMessage $message): void
    {
        $email = MessageConverter::toEmail($message->getOriginalMessage());

        $this->client->messages->send(['message' => [
            'from_email' => $email->getFrom(),
            'to' => collect($email->getTo())->map(function ($email) {
                return ['email' => $email->getAddress(), 'type' => 'to'];
            })->all(),
            'subject' => $email->getSubject(),
            'text' => $email->getTextBody(),
        ]]);
    }

    /**
     * Get the string representation of the transport.
     *
     * @return string
     */
    public function __toString(): string
    {
        return 'mailchimp';
    }
}
```

Khi bạn đã xác định phương tiện truyền tải tùy chỉnh của mình, bạn có thể đăng ký nó thông qua phương thức mở rộng do facade Mail cung cấp. Thông thường, điều này phải được thực hiện trong phương thức `boot` của `AppServiceProvider` của ứng dụng của bạn. Đối số `$config` sẽ được truyền vào hàm nặc danh được khai báo trên phương thức extend. Đối số này sẽ chứa mảng cấu hình được xác định cho mailer trong tập tin cấu hình `config/mail.php` của ứng dụng:

```
use App\Mail\MailchimpTransport;
use Illuminate\Support\Facades\Mail;

/**
```

```
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
      Mail::extend('mailchimp', function (array $config = []) {
        return new MailchimpTransport(/* ... */);
      })
    }
```

Khi cơ chế truyền tải tự tạo của bạn đã được tạo và đăng ký, bạn có thể tạo một mailer trong tập tin cấu hình `config/mail.php` của ứng dụng sử dụng phương tiện truyền tải mới:

```
'mailchimp' => [
  'transport' => 'mailchimp',
  // ...
],
```

Cơ chế truyền tải bổ sung với Symfony

Laravel có hỗ trợ một số phương thức truyền tải mail được Symfony duy trì hiện tại như Mailgun và Postmark. Tuy nhiên, bạn có thể muốn mở rộng Laravel với sự hỗ trợ cho các phương tiện truyền tải bổ sung được Symfony bảo trì. Bạn có thể làm như vậy bằng cách dùng lệnh Composer và đăng ký cơ thế truyền tải bổ sung với Laravel. Ví dụ: bạn có thể cài đặt và đăng ký mailer Symfony "Sendinblue":

```
composer require symfony/sendinblue-mailer
```

Khi Sendinblue đã được cài đặt, bạn có thể thêm một mục nhập cho thông tin xác thực API Sendinblue của mình vào tập tin cấu hình dịch vụ của ứng dụng:

```
'sendinblue' => [
  'key' => 'your-api-key',
],
```

Cuối cùng, bạn có thể sử dụng phương thức extend của facade `Mail` để đăng ký cơ chế truyền tải với Laravel. Thông thường, điều này phải được thực hiện qua phương thức `boot` của nhà cung cấp dịch vụ:

```
use Illuminate\Support\Facades\Mail;
use Symfony\Component\Mailer\Bridge\Sendinblue\Transport\SendinblueTransportFactory;
use Symfony\Component\Mailer\Transport\Dsn;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
  Mail::extend('sendinblue', function () {
    return (new SendinblueTransportFactory)->create(
      new Dsn(
        'sendinblue+api',
        'default',
        config('services.sendinblue.key')
      )
    );
  });
}
```