Mail

Introduction

Sending email doesn't have to be complicated. Laravel provides a clean, simple email API powered by the popular Symfony Mailer component. Laravel and Symfony Mailer provide drivers for sending email via SMTP, Mailgun, Postmark, Resend, Amazon SES, and `sendmail`, allowing you to quickly get started sending mail through a local or cloud-based service of your choice.

Configuration

Laravel's email services may be configured via your application's `config/mail.php` configuration file. Each mailer configured within this file may have its own unique configuration and even its own unique "transport", allowing your application to use different email services to send certain email messages. For example, your application might use Postmark to send transactional emails while using Amazon SES to send bulk emails.

Within your `mail` configuration file, you will find a `mailers` configuration array. This array contains a sample configuration entry for each of the major mail drivers / transports supported by Laravel, while the `default` configuration value determines which mailer will be used by default when your application needs to send an email message.

Driver / Transport Prerequisites

The API based drivers such as Mailgun, Postmark, and Resend are often simpler and faster than sending mail via SMTP servers. Whenever possible, we recommend that you use one of these drivers.

Mailgun Driver

To use the Mailgun driver, install Symfony's Mailgun Mailer transport via Composer:

```
composer require symfony/mailgun-mailer symfony/http-client
```

Next, you will need to make two changes in your application's `config/mail.php` configuration file. First, set your default mailer to `mailgun`:

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

Second, add the following configuration array to your array of `mailers`:

```
'mailgun' => [
    'transport' => 'mailgun',
    // 'client' => [
    //     'timeout' => 5,
    // ],
],
```

After configuring your application's default mailer, add the following options to your `config/services.php` configuration file:

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

```
    ],
```

If you are not using the United States Mailgun region, you may define your region's endpoint in the `services` configuration file:

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

Postmark Driver

To use the Postmark driver, install Symfony's Postmark Mailer transport via Composer:

```
composer require symfony/postmark-mailer symfony/http-client
```

Next, set the `default` option in your application's `config/mail.php` configuration file to `postmark`. After configuring your application's default mailer, ensure that your `config/services.php` configuration file contains the following options:

```
    'postmark' => [
        'key' => env('POSTMARK_API_KEY'),
    ],
```

If you would like to specify the Postmark message stream that should be used by a given mailer, you may add the `message_stream_id` configuration option to the mailer's configuration array. This configuration array can be found in your application's `config/mail.php` configuration file:

```
    'postmark' => [
```

```
        'transport' => 'postmark',
        'message_stream_id' => env('POSTMARK_MESSAGE_STREAM_ID'),
        // 'client' => [
        //     'timeout' => 5,
        // ],
    ],
```

This way you are also able to set up multiple Postmark mailers with different message streams.

Resend Driver

To use the Resend driver, install Resend's PHP SDK via Composer:

```
composer require resend/resend-php
```

Next, set the `default` option in your application's `config/mail.php` configuration file to `resend`. After configuring your application's default mailer, ensure that your `config/services.php` configuration file contains the following options:

```
    'resend' => [
        'key' => env('RESEND_API_KEY'),
    ],
```

SES Driver

To use the Amazon SES driver you must first install the Amazon AWS SDK for PHP. You may install this library via the Composer package manager:

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

Next, set the `default` option in your `config/mail.php` configuration file to `ses` and verify that your `config/services.php` configuration file contains the following options:

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

To utilize AWS temporary credentials via a session token, you may add a `token` key to your application's SES configuration:

```
    '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'),
    ],
```

To interact with SES's subscription management features, you may return the `X-Ses-List-Management-Options` header in the array returned by the headers method of a mail message:

```
    /**
     * Get the message headers.
     */
    public function headers(): Headers
    {
        return new Headers(
            text: [
                'X-Ses-List-Management-Options' => 'contactListName=MyContact
            ],
        );
    }
```

If you would like to define additional options that Laravel should pass to the AWS SDK's `SendEmail` method when sending an email, you may define an `options` array within

your `ses` configuration:

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

Failover Configuration

Sometimes, an external service you have configured to send your application's mail may be down. In these cases, it can be useful to define one or more backup mail delivery configurations that will be used in case your primary delivery driver is down.

To accomplish this, you should define a mailer within your application's `mail` configuration file that uses the `failover` transport. The configuration array for your application's `failover` mailer should contain an array of `mailers` that reference the order in which configured mailers should be chosen for delivery:

```
'mailers' => [
    'failover' => [
        'transport' => 'failover',
        'mailers' => [
            'postmark',
            'mailgun',
            'sendmail',
        ],
        'retry_after' => 60,
    ],

    // ...
```

```
    ],
```

Once you have configured a mailer that uses the `failover` transport, you will need to set the failover mailer as your default mailer in your application's `.env` file to make use of the failover functionality:

```
MAIL_MAILER=failover
```

Round Robin Configuration

The `roundrobin` transport allows you to distribute your mailing workload across multiple mailers. To get started, define a mailer within your application's `mail` configuration file that uses the `roundrobin` transport. The configuration array for your application's `roundrobin` mailer should contain an array of `mailers` that reference which configured mailers should be used for delivery:

```
    'mailers' => [
        'roundrobin' => [
            'transport' => 'roundrobin',
            'mailers' => [
                'ses',
                'postmark',
            ],
            'retry_after' => 60,
        ],

        // ...
    ],
```

Once your round robin mailer has been defined, you should set this mailer as the default mailer used by your application by specifying its name as the value of the `default` configuration key within your application's `mail` configuration file:

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

The round robin transport selects a random mailer from the list of configured mailers and then switches to the next available mailer for each subsequent email. In contrast to `failover` transport, which helps to achieve *high availability*, the `roundrobin` transport provides *load balancing*.

Generating Mailables

When building Laravel applications, each type of email sent by your application is represented as a "mailable" class. These classes are stored in the `app/Mail` directory. Don't worry if you don't see this directory in your application, since it will be generated for you when you create your first mailable class using the `make:mail` Artisan command:

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

Writing Mailables

Once you have generated a mailable class, open it up so we can explore its contents. Mailable class configuration is done in several methods, including the `envelope`, `content`, and `attachments` methods.

The `envelope` method returns an `Illuminate\Mail\Mailables\Envelope` object that defines the subject and, sometimes, the recipients of the message. The `content` method returns an `Illuminate\Mail\Mailables\Content` object that defines the Blade template that will be used to generate the message content.

Configuring the Sender

Using the Envelope

First, let's explore configuring the sender of the email. Or, in other words, who the email is going to be "from". There are two ways to configure the sender. First, you may specify the "from" address on your message's envelope:

```
use Illuminate\Mail\Mailables\Address;
use Illuminate\Mail\Mailables\Envelope;

/**
 * Get the message envelope.
 */
public function envelope(): Envelope
{
    return new Envelope(
        from: new Address('jeffrey@example.com', 'Jeffrey Way'),
        subject: 'Order Shipped',
    );
}
```

If you would like, you may also specify a `replyTo` address:

```
return new Envelope(
    from: new Address('jeffrey@example.com', 'Jeffrey Way'),
    replyTo: [
        new Address('taylor@example.com', 'Taylor Otwell'),
    ],
    subject: 'Order Shipped',
);
```

Using a Global `from` Address

However, if your application uses the same "from" address for all of its emails, it can become cumbersome to add it to each mailable class you generate. Instead, you may specify a global "from" address in your `config/mail.php` configuration file. This address will be used if no other "from" address is specified within the mailable class:

```
'from' => [
```

```
        'address' => env('MAIL_FROM_ADDRESS', 'hello@example.com'),
        'name' => env('MAIL_FROM_NAME', 'Example'),
    ],
```

In addition, you may define a global "reply_to" address within your `config/mail.php` configuration file:

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

Configuring the View

Within a mailable class's `content` method, you may define the `view`, or which template should be used when rendering the email's contents. Since each email typically uses a Blade template to render its contents, you have the full power and convenience of the Blade templating engine when building your email's HTML:

```
    /**
     * Get the message content definition.
     */
    public function content(): Content
    {
        return new Content(
            view: 'mail.orders.shipped',
        );
    }
```

You may wish to create a `resources/views/mail` directory to house all of your email templates; however, you are free to place them wherever you wish within your `resources/views` directory.

Plain Text Emails

If you would like to define a plain-text version of your email, you may specify the plain-text template when creating the message's `Content` definition. Like the `view` parameter, the `text` parameter should be a template name which will be used to render the contents of the email. You are free to define both an HTML and plain-text version of your message:

```
/**
 * Get the message content definition.
 */
public function content(): Content
{
    return new Content(
        view: 'mail.orders.shipped',
        text: 'mail.orders.shipped-text'
    );
}
```

For clarity, the `html` parameter may be used as an alias of the `view` parameter:

```
return new Content(
    html: 'mail.orders.shipped',
    text: 'mail.orders.shipped-text'
);
```

View Data

Via Public Properties

Typically, you will want to pass some data to your view that you can utilize when rendering the email's HTML. There are two ways you may make data available to your view. First, any public property defined on your mailable class will automatically be made available to the view. So, for example, you may pass data into your mailable class's constructor and set that data to public properties defined on the class:

```php
<?php

namespace App\Mail;

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

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * Create a new message instance.
     */
    public function __construct(
        public Order $order,
    ) {}

    /**
     * Get the message content definition.
     */
    public function content(): Content
    {
        return new Content(
            view: 'mail.orders.shipped',
        );
    }
}
```

Once the data has been set to a public property, it will automatically be available in your view, so you may access it like you would access any other data in your Blade templates:

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

Via the `with` Parameter:

If you would like to customize the format of your email's data before it is sent to the template, you may manually pass your data to the view via the `Content` definition's `with` parameter. Typically, you will still pass data via the mailable class's constructor; however, you should set this data to `protected` or `private` properties so the data is not automatically made available to the template:

```php
<?php

namespace App\Mail;

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

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * Create a new message instance.
     */
    public function __construct(
        protected Order $order,
    ) {}

    /**
     * Get the message content definition.
     */
    public function content(): Content
    {
        return new Content(
            view: 'mail.orders.shipped',
            with: [
                'orderName' => $this->order->name,
                'orderPrice' => $this->order->price,
```

```
            ],
        );
    }
}
```

Once the data has been passed via the `with` parameter, it will automatically be available in your view, so you may access it like you would access any other data in your Blade templates:

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

Attachments

To add attachments to an email, you will add attachments to the array returned by the message's `attachments` method. First, you may add an attachment by providing a file path to the `fromPath` method provided by the `Attachment` class:

```
use Illuminate\Mail\Mailables\Attachment;

/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromPath('/path/to/file'),
    ];
}
```

When attaching files to a message, you may also specify the display name and / or MIME type for the attachment using the `as` and `withMime` methods:

```php
/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromPath('/path/to/file')
            ->as('name.pdf')
            ->withMime('application/pdf'),
    ];
}
```

Attaching Files From Disk

If you have stored a file on one of your filesystem disks, you may attach it to the email using the `fromStorage` attachment method:

```php
/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
 */
public function attachments(): array
{
    return [
        Attachment::fromStorage('/path/to/file'),
    ];
}
```

Of course, you may also specify the attachment's name and MIME type:

```php
/**
 * Get the attachments for the message.
 *
 * @return array<int, \Illuminate\Mail\Mailables\Attachment>
```

```
     */
    public function attachments(): array
    {
        return [
            Attachment::fromStorage('/path/to/file')
                ->as('name.pdf')
                ->withMime('application/pdf'),
        ];
    }
```

The `fromStorageDisk` method may be used if you need to specify a storage disk other than your default disk:

```
    /**
     * Get the attachments for the message.
     *
     * @return array<int, \Illuminate\Mail\Mailables\Attachment>
     */
    public function attachments(): array
    {
        return [
            Attachment::fromStorageDisk('s3', '/path/to/file')
                ->as('name.pdf')
                ->withMime('application/pdf'),
        ];
    }
```

Raw Data Attachments

The `fromData` attachment method may be used to attach a raw string of bytes as an attachment. For example, you might use this method if you have generated a PDF in memory and want to attach it to the email without writing it to disk. The `fromData` method accepts a closure which resolves the raw data bytes as well as the name that the attachment should be assigned:

```
    /**
     * Get the attachments for the message.
```

```
     *
     * @return array<int, \Illuminate\Mail\Mailables\Attachment>
     */
    public function attachments(): array
    {
        return [
            Attachment::fromData(fn () => $this->pdf, 'Report.pdf')
                ->withMime('application/pdf'),
        ];
    }
```

Inline Attachments

Embedding inline images into your emails is typically cumbersome; however, Laravel provides a convenient way to attach images to your emails. To embed an inline image, use the `embed` method on the `$message` variable within your email template. Laravel automatically makes the `$message` variable available to all of your email templates, so you don't need to worry about passing it in manually:

```
<body>
    Here is an image:

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

The `$message` variable is not available in plain-text message templates since plain-text messages do not utilize inline attachments.

Embedding Raw Data Attachments

If you already have a raw image data string you wish to embed into an email template, you may call the `embedData` method on the `$message` variable. When calling the

`embedData` method, you will need to provide a filename that should be assigned to the embedded image:

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

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

Attachable Objects

While attaching files to messages via simple string paths is often sufficient, in many cases the attachable entities within your application are represented by classes. For example, if your application is attaching a photo to a message, your application may also have a `Photo` model that represents that photo. When that is the case, wouldn't it be convenient to simply pass the `Photo` model to the `attach` method? Attachable objects allow you to do just that.

To get started, implement the `Illuminate\Contracts\Mail\Attachable` interface on the object that will be attachable to messages. This interface dictates that your class defines a `toMailAttachment` method that returns an `Illuminate\Mail\Attachment` instance:

```
<?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.
     */
    public function toMailAttachment(): Attachment
```

```
        {
            return Attachment::fromPath('/path/to/file');
        }
    }
```

Once you have defined your attachable object, you may return an instance of that object from the `attachments` method when building an email message:

```
    /**
     * Get the attachments for the message.
     *
     * @return array<int, \Illuminate\Mail\Mailables\Attachment>
     */
    public function attachments(): array
    {
        return [$this->photo];
    }
```

Of course, attachment data may be stored on a remote file storage service such as Amazon S3. So, Laravel also allows you to generate attachment instances from data that is stored on one of your application's filesystem disks:

```
    // 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);
```

In addition, you may create attachment instances via data that you have in memory. To accomplish this, provide a closure to the `fromData` method. The closure should return the raw data that represents the attachment:

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

Laravel also provides additional methods that you may use to customize your attachments. For example, you may use the `as` and `withMime` methods to customize the file's name and MIME type:

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

Headers

Sometimes you may need to attach additional headers to the outgoing message. For instance, you may need to set a custom `Message-Id` or other arbitrary text headers.

To accomplish this, define a `headers` method on your mailable. The `headers` method should return an `Illuminate\Mail\Mailables\Headers` instance. This class accepts `messageId`, `references`, and `text` parameters. Of course, you may provide only the parameters you need for your particular message:

```
use Illuminate\Mail\Mailables\Headers;

/**
 * Get the message headers.
 */
public function headers(): Headers
{
    return new Headers(
        messageId: 'custom-message-id@example.com',
        references: ['previous-message@example.com'],
        text: [
            'X-Custom-Header' => 'Custom Value',
        ],
    );
}
```

Tags and Metadata

Some third-party email providers such as Mailgun and Postmark support message "tags" and "metadata", which may be used to group and track emails sent by your application. You may add tags and metadata to an email message via your `Envelope` definition:

```php
use Illuminate\Mail\Mailables\Envelope;

/**
 * Get the message envelope.
 *
 * @return \Illuminate\Mail\Mailables\Envelope
 */
public function envelope(): Envelope
{
    return new Envelope(
        subject: 'Order Shipped',
        tags: ['shipment'],
        metadata: [
            'order_id' => $this->order->id,
        ],
    );
}
```

If your application is using the Mailgun driver, you may consult Mailgun's documentation for more information on tags and metadata. Likewise, the Postmark documentation may also be consulted for more information on their support for tags and metadata.

If your application is using Amazon SES to send emails, you should use the `metadata` method to attach SES "tags" to the message.

Customizing the Symfony Message

Laravel's mail capabilities are powered by Symfony Mailer. Laravel allows you to register custom callbacks that will be invoked with the Symfony Message instance before sending the message. This gives you an opportunity to deeply customize the

message before it is sent. To accomplish this, define a `using` parameter on your `Envelope` definition:

```php
use Illuminate\Mail\Mailables\Envelope;
use Symfony\Component\Mime\Email;

/**
 * Get the message envelope.
 */
public function envelope(): Envelope
{
    return new Envelope(
        subject: 'Order Shipped',
        using: [
            function (Email $message) {
                // ...
            },
        ]
    );
}
```

Markdown Mailables

Markdown mailable messages allow you to take advantage of the pre-built templates and components of mail notifications in your mailables. Since the messages are written in Markdown, Laravel is able to render beautiful, responsive HTML templates for the messages while also automatically generating a plain-text counterpart.

Generating Markdown Mailables

To generate a mailable with a corresponding Markdown template, you may use the `--markdown` option of the `make:mail` Artisan command:

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

Then, when configuring the mailable `Content` definition within its `content` method, use the `markdown` parameter instead of the `view` parameter:

```php
use Illuminate\Mail\Mailables\Content;

/**
 * Get the message content definition.
 */
public function content(): Content
{
    return new Content(
        markdown: 'mail.orders.shipped',
        with: [
            'url' => $this->orderUrl,
        ],
    );
}
```

Writing Markdown Messages

Markdown mailables use a combination of Blade components and Markdown syntax which allow you to easily construct mail messages while leveraging Laravel's pre-built email UI components:

```blade
<x-mail::message>
# Order Shipped

Your order has been shipped!

<x-mail::button :url="$url">
View Order
</x-mail::button>

Thanks,<br>
{{ config('app.name') }}
</x-mail::message>
```

Do not use excess indentation when writing Markdown emails. Per Markdown standards, Markdown parsers will render indented content as code blocks.

Button Component

The button component renders a centered button link. The component accepts two arguments, a `url` and an optional `color`. Supported colors are `primary`, `success`, and `error`. You may add as many button components to a message as you wish:

```
<x-mail::button :url="$url" color="success">
View Order
</x-mail::button>
```

Panel Component

The panel component renders the given block of text in a panel that has a slightly different background color than the rest of the message. This allows you to draw attention to a given block of text:

```
<x-mail::panel>
This is the panel content.
</x-mail::panel>
```

Table Component

The table component allows you to transform a Markdown table into an HTML table. The component accepts the Markdown table as its content. Table column alignment is supported using the default Markdown table alignment syntax:

```
<x-mail::table>
| Laravel       | Table         | Example       |
| ------------- | :-----------: | ------------: |
```

```
| Col 2 is      | Centered      | $10           |
| Col 3 is      | Right-Aligned | $20           |
</x-mail::table>
```

Customizing the Components

You may export all of the Markdown mail components to your own application for customization. To export the components, use the `vendor:publish` Artisan command to publish the `laravel-mail` asset tag:

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

This command will publish the Markdown mail components to the `resources/views/vendor/mail` directory. The `mail` directory will contain an `html` and a `text` directory, each containing their respective representations of every available component. You are free to customize these components however you like.

Customizing the CSS

After exporting the components, the `resources/views/vendor/mail/html/themes` directory will contain a `default.css` file. You may customize the CSS in this file and your styles will automatically be converted to inline CSS styles within the HTML representations of your Markdown mail messages.

If you would like to build an entirely new theme for Laravel's Markdown components, you may place a CSS file within the `html/themes` directory. After naming and saving your CSS file, update the `theme` option of your application's `config/mail.php` configuration file to match the name of your new theme.

To customize the theme for an individual mailable, you may set the `$theme` property of the mailable class to the name of the theme that should be used when sending that mailable.

Sending Mail

To send a message, use the `to` method on the `Mail` facade. The `to` method accepts an email address, a user instance, or a collection of users. If you pass an object or collection of objects, the mailer will automatically use their `email` and `name` properties when determining the email's recipients, so make sure these attributes are available on your objects. Once you have specified your recipients, you may pass an instance of your mailable class to the `send` method:

```php
<?php

namespace App\Http\Controllers;

use App\Mail\OrderShipped;
use App\Models\Order;
use Illuminate\Http\RedirectResponse;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;

class OrderShipmentController extends Controller
{
    /**
     * Ship the given order.
     */
    public function store(Request $request): RedirectResponse
    {
        $order = Order::findOrFail($request->order_id);

        // Ship the order...

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

        return redirect('/orders');
    }
}
```

You are not limited to just specifying the "to" recipients when sending a message. You are free to set "to", "cc", and "bcc" recipients by chaining their respective methods together:

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

Looping Over Recipients

Occasionally, you may need to send a mailable to a list of recipients by iterating over an array of recipients / email addresses. However, since the `to` method appends email addresses to the mailable's list of recipients, each iteration through the loop will send another email to every previous recipient. Therefore, you should always re-create the mailable instance for each recipient:

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

Sending Mail via a Specific Mailer

By default, Laravel will send email using the mailer configured as the `default` mailer in your application's `mail` configuration file. However, you may use the `mailer` method to send a message using a specific mailer configuration:

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

Queueing Mail

Queueing a Mail Message

Since sending email messages can negatively impact the response time of your application, many developers choose to queue email messages for background sending. Laravel makes this easy using its built-in unified queue API. To queue a mail

message, use the `queue` method on the `Mail` facade after specifying the message's recipients:

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

This method will automatically take care of pushing a job onto the queue so the message is sent in the background. You will need to configure your queues before using this feature.

Delayed Message Queueing

If you wish to delay the delivery of a queued email message, you may use the `later` method. As its first argument, the `later` method accepts a `DateTime` instance indicating when the message should be sent:

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

Pushing to Specific Queues

Since all mailable classes generated using the `make:mail` command make use of the `Illuminate\Bus\Queueable` trait, you may call the `onQueue` and `onConnection` methods on any mailable class instance, allowing you to specify the connection and queue name for the message:

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

Mail::to($request->user())
```

```
        ->cc($moreUsers)
        ->bcc($evenMoreUsers)
        ->queue($message);
```

Queueing by Default

If you have mailable classes that you want to always be queued, you may implement the `ShouldQueue` contract on the class. Now, even if you call the `send` method when mailing, the mailable will still be queued since it implements the contract:

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

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

Queued Mailables and Database Transactions

When queued mailables are dispatched within database transactions, they may be processed by the queue before the database transaction has committed. When this happens, any updates you have made to models or database records during the database transaction may not yet be reflected in the database. In addition, any models or database records created within the transaction may not exist in the database. If your mailable depends on these models, unexpected errors can occur when the job that sends the queued mailable is processed.

If your queue connection's `after_commit` configuration option is set to `false`, you may still indicate that a particular queued mailable should be dispatched after all open database transactions have been committed by calling the `afterCommit` method when sending the mail message:

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

Alternatively, you may call the `afterCommit` method from your mailable's constructor:

```php
<?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.
     */
    public function __construct()
    {
        $this->afterCommit();
    }
}
```

> To learn more about working around these issues, please review the documentation regarding queued jobs and database transactions.

Queued Email Failures

When a queued email fails, the `failed` method on the queued mailable class will be invoked if it has been defined. The `Throwable` instance that caused the queued email to fail will be passed to the `failed` method:

```php
<?php

```

```php
namespace App\Mail;

use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Throwable;

class OrderDelayed extends Mailable implements ShouldQueue
{
    use SerializesModels;

    /**
     * Handle a queued email's failure.
     */
    public function failed(Throwable $exception): void
    {
        // ...
    }
}
```

Rendering Mailables

Sometimes you may wish to capture the HTML content of a mailable without sending it. To accomplish this, you may call the `render` method of the mailable. This method will return the evaluated HTML content of the mailable as a string:

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

$invoice = Invoice::find(1);

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

Previewing Mailables in the Browser

When designing a mailable's template, it is convenient to quickly preview the rendered mailable in your browser like a typical Blade template. For this reason, Laravel allows you to return any mailable directly from a route closure or controller. When a mailable is returned, it will be rendered and displayed in the browser, allowing you to quickly preview its design without needing to send it to an actual email address:

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

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

Localizing Mailables

Laravel allows you to send mailables in a locale other than the request's current locale, and will even remember this locale if the mail is queued.

To accomplish this, the `Mail` facade offers a `locale` method to set the desired language. The application will change into this locale when the mailable's template is being evaluated and then revert back to the previous locale when evaluation is complete:

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

User Preferred Locales

Sometimes, applications store each user's preferred locale. By implementing the `HasLocalePreference` contract on one or more of your models, you may instruct Laravel to use this stored locale when sending mail:

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

```php

class User extends Model implements HasLocalePreference
{
    /**
     * Get the user's preferred locale.
     */
    public function preferredLocale(): string
    {
        return $this->locale;
    }
}
```

Once you have implemented the interface, Laravel will automatically use the preferred locale when sending mailables and notifications to the model. Therefore, there is no need to call the `locale` method when using this interface:

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

Testing

Testing Mailable Content

Laravel provides a variety of methods for inspecting your mailable's structure. In addition, Laravel provides several convenient methods for testing that your mailable contains the content that you expect:

Pest PHPUnit

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

test('mailable content', function () {
    $user = User::factory()->create();

    $mailable = new InvoicePaid($user);
```

```
8
9           $mailable->assertFrom('jeffrey@example.com');
10          $mailable->assertTo('taylor@example.com');
11          $mailable->assertHasCc('abigail@example.com');
12          $mailable->assertHasBcc('victoria@example.com');
13          $mailable->assertHasReplyTo('tyler@example.com');
14          $mailable->assertHasSubject('Invoice Paid');
15          $mailable->assertHasTag('example-tag');
16          $mailable->assertHasMetadata('key', 'value');
17
18          $mailable->assertSeeInHtml($user->email);
19          $mailable->assertDontSeeInHtml('Invoice Not Paid');
20          $mailable->assertSeeInOrderInHtml(['Invoice Paid', 'Thanks']);
21
22          $mailable->assertSeeInText($user->email);
23          $mailable->assertDontSeeInText('Invoice Not Paid');
24          $mailable->assertSeeInOrderInText(['Invoice Paid', 'Thanks']);
25
26          $mailable->assertHasAttachment('/path/to/file');
27          $mailable->assertHasAttachment(Attachment::fromPath('/path/to/file'))
28          $mailable->assertHasAttachedData($pdfData, 'name.pdf', ['mime' => 'ap
29          $mailable->assertHasAttachmentFromStorage('/path/to/file', 'name.pdf'
30          $mailable->assertHasAttachmentFromStorageDisk('s3', '/path/to/file',
31      });
```

As you might expect, the "HTML" assertions assert that the HTML version of your mailable contains a given string, while the "text" assertions assert that the plain-text version of your mailable contains a given string.

Testing Mailable Sending

We suggest testing the content of your mailables separately from your tests that assert that a given mailable was "sent" to a specific user. Typically, the content of mailables is not relevant to the code you are testing, and it is sufficient to simply assert that Laravel was instructed to send a given mailable.

You may use the `Mail` facade's `fake` method to prevent mail from being sent. After calling the `Mail` facade's `fake` method, you may then assert that mailables were instructed to be sent to users and even inspect the data the mailables received:

Pest PHPUnit

```php
<?php

use App\Mail\OrderShipped;
use Illuminate\Support\Facades\Mail;

test('orders can be shipped', function () {
    Mail::fake();

    // Perform order shipping...

    // Assert that no mailables were sent...
    Mail::assertNothingSent();

    // Assert that a mailable was sent...
    Mail::assertSent(OrderShipped::class);

    // Assert a mailable was sent twice...
    Mail::assertSent(OrderShipped::class, 2);

    // Assert a mailable was sent to an email address...
    Mail::assertSent(OrderShipped::class, 'example@laravel.com');

    // Assert a mailable was sent to multiple email addresses...
    Mail::assertSent(OrderShipped::class, ['example@laravel.com', '...']);

    // Assert a mailable was not sent...
    Mail::assertNotSent(AnotherMailable::class);

    // Assert a mailable was sent twice...
    Mail::assertSentTimes(OrderShipped::class, 2);

    // Assert 3 total mailables were sent...
    Mail::assertSentCount(3);
});
```

If you are queueing mailables for delivery in the background, you should use the `assertQueued` method instead of `assertSent`:

```php
Mail::assertQueued(OrderShipped::class);
Mail::assertNotQueued(OrderShipped::class);
Mail::assertNothingQueued();
Mail::assertQueuedCount(3);
```

You can also assert the total number of mailables that have been sent or queued using the `assertOutgoingCount` method:

```php
Mail::assertOutgoingCount(3);
```

You may pass a closure to the `assertSent`, `assertNotSent`, `assertQueued`, or `assertNotQueued` methods in order to assert that a mailable was sent that passes a given "truth test". If at least one mailable was sent that passes the given truth test then the assertion will be successful:

```php
Mail::assertSent(function (OrderShipped $mail) use ($order) {
    return $mail->order->id === $order->id;
});
```

When calling the `Mail` facade's assertion methods, the mailable instance accepted by the provided closure exposes helpful methods for examining the mailable:

```php
Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) use ($
    return $mail->hasTo($user->email) &&
           $mail->hasCc('...') &&
           $mail->hasBcc('...') &&
           $mail->hasReplyTo('...') &&
           $mail->hasFrom('...') &&
           $mail->hasSubject('...') &&
           $mail->hasMetadata('order_id', $mail->order->id);
           $mail->usesMailer('ses');
});
```

The mailable instance also includes several helpful methods for examining the attachments on a mailable:

```
use Illuminate\Mail\Mailables\Attachment;

Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) {
    return $mail->hasAttachment(
        Attachment::fromPath('/path/to/file')
            ->as('name.pdf')
            ->withMime('application/pdf')
    );
});

Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) {
    return $mail->hasAttachment(
        Attachment::fromStorageDisk('s3', '/path/to/file')
    );
});

Mail::assertSent(OrderShipped::class, function (OrderShipped $mail) use ($
    return $mail->hasAttachment(
        Attachment::fromData(fn () => $pdfData, 'name.pdf')
    );
});
```

You may have noticed that there are two methods for asserting that mail was not sent: `assertNotSent` and `assertNotQueued`. Sometimes you may wish to assert that no mail was sent **or** queued. To accomplish this, you may use the `assertNothingOutgoing` and `assertNotOutgoing` methods:

```
Mail::assertNothingOutgoing();

Mail::assertNotOutgoing(function (OrderShipped $mail) use ($order) {
    return $mail->order->id === $order->id;
```

```
});
```

Mail and Local Development

When developing an application that sends email, you probably don't want to actually send emails to live email addresses. Laravel provides several ways to "disable" the actual sending of emails during local development.

Log Driver

Instead of sending your emails, the `log` mail driver will write all email messages to your log files for inspection. Typically, this driver would only be used during local development. For more information on configuring your application per environment, check out the configuration documentation.

HELO / Mailtrap / Mailpit

Alternatively, you may use a service like HELO or Mailtrap and the `smtp` driver to send your email messages to a "dummy" mailbox where you may view them in a true email client. This approach has the benefit of allowing you to actually inspect the final emails in Mailtrap's message viewer.

If you are using Laravel Sail, you may preview your messages using Mailpit. When Sail is running, you may access the Mailpit interface at: `http://localhost:8025`.

Using a Global `to` Address

Finally, you may specify a global "to" address by invoking the `alwaysTo` method offered by the `Mail` facade. Typically, this method should be called from the `boot` method of one of your application's service providers:

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

/**
 * Bootstrap any application services.
 */
public function boot(): void
```

```
    {
        if ($this->app->environment('local')) {
            Mail::alwaysTo('taylor@example.com');
        }
    }
```

When using the `alwaysTo` method, any additional "cc" or "bcc" addresses on mail messages will be removed.

Events

Laravel dispatches two events while sending mail messages. The `MessageSending` event is dispatched prior to a message being sent, while the `MessageSent` event is dispatched after a message has been sent. Remember, these events are dispatched when the mail is being *sent*, not when it is queued. You may create event listeners for these events within your application:

```
use Illuminate\Mail\Events\MessageSending;
// use Illuminate\Mail\Events\MessageSent;

class LogMessage
{
    /**
     * Handle the event.
     */
    public function handle(MessageSending $event): void
    {
        // ...
    }
}
```

Custom Transports

Laravel includes a variety of mail transports; however, you may wish to write your own transports to deliver email via other services that Laravel does not support out of the

box. To get started, define a class that extends the `Symfony\Component\Mailer\Transport\AbstractTransport` class. Then, implement the `doSend` and `__toString` methods on your transport:

```php
<?php

namespace App\Mail;

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

class MailchimpTransport extends AbstractTransport
{
    /**
     * Create a new Mailchimp transport instance.
     */
    public function __construct(
        protected ApiClient $client,
    ) {
        parent::__construct();
    }

    /**
     * {@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 (Address $emai
                return ['email' => $email->getAddress(), 'type' => 'to'];
            })->all(),
            'subject' => $email->getSubject(),
            'text' => $email->getTextBody(),
        ]]);
    }
```

```
38
39          /**
40           * Get the string representation of the transport.
41           */
42          public function __toString(): string
43          {
44              return 'mailchimp';
45          }
46      }
```

Once you've defined your custom transport, you may register it via the `extend` method provided by the `Mail` facade. Typically, this should be done within the `boot` method of your application's `AppServiceProvider`. A `$config` argument will be passed to the closure provided to the `extend` method. This argument will contain the configuration array defined for the mailer in the application's `config/mail.php` configuration file:

```
1   use App\Mail\MailchimpTransport;
2   use Illuminate\Support\Facades\Mail;
3   use MailchimpTransactional\ApiClient;
4
5   /**
6    * Bootstrap any application services.
7    */
8   public function boot(): void
9   {
10      Mail::extend('mailchimp', function (array $config = []) {
11          $client = new ApiClient;
12
13          $client->setApiKey($config['key']);
14
15          return new MailchimpTransport($client);
16      });
17  }
```

Once your custom transport has been defined and registered, you may create a mailer definition within your application's `config/mail.php` configuration file that utilizes the new transport:

```
'mailchimp' => [
    'transport' => 'mailchimp',
    'key' => env('MAILCHIMP_API_KEY'),
    // ...
],
```

Additional Symfony Transports

Laravel includes support for some existing Symfony maintained mail transports like Mailgun and Postmark. However, you may wish to extend Laravel with support for additional Symfony maintained transports. You can do so by requiring the necessary Symfony mailer via Composer and registering the transport with Laravel. For example, you may install and register the "Brevo" (formerly "Sendinblue") Symfony mailer:

```
composer require symfony/brevo-mailer symfony/http-client
```

Once the Brevo mailer package has been installed, you may add an entry for your Brevo API credentials to your application's `services` configuration file:

```
'brevo' => [
    'key' => env('BREVO_API_KEY'),
],
```

Next, you may use the `Mail` facade's `extend` method to register the transport with Laravel. Typically, this should be done within the `boot` method of a service provider:

```
use Illuminate\Support\Facades\Mail;
use Symfony\Component\Mailer\Bridge\Brevo\Transport\BrevoTransportFactory
use Symfony\Component\Mailer\Transport\Dsn;

/**
 * Bootstrap any application services.
 */
```

```php
    public function boot(): void
    {
        Mail::extend('brevo', function () {
            return (new BrevoTransportFactory)->create(
                new Dsn(
                    'brevo+api',
                    'default',
                    config('services.brevo.key')
                )
            );
        });
    }
```

Once your transport has been registered, you may create a mailer definition within your application's `config/mail.php` configuration file that utilizes the new transport:

```php
    'brevo' => [
        'transport' => 'brevo',
        // ...
    ],
```