Mail

Introduction

Laravel provides a clean, simple API over the popular SwiftMailer library with drivers for SMTP, Mailgun, Postmark, Amazon SES, and `sendmail`, allowing you to quickly get started sending mail through a local or cloud based service of your choice.

Driver Prerequisites

The API based drivers such as Mailgun and Postmark are often simpler and faster than SMTP servers. If possible, you should use one of these drivers. All of the API drivers require the Guzzle HTTP library, which may be installed via the Composer package manager:

```
composer require guzzlehttp/guzzle
```

Mailgun Driver

To use the Mailgun driver, first install Guzzle, then set the `driver` option in your `config/mail.php` configuration file to `mailgun`. Next, verify that your `config/services.php` configuration file contains the following options:

```
'mailgun' => [
    'domain' => 'your-mailgun-domain',
    'secret' => 'your-mailgun-key',
],
```

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

```
'mailgun' => [
    'domain' => 'your-mailgun-domain',
    'secret' => 'your-mailgun-key',
    'endpoint' => 'api.eu.mailgun.net',
],
```

Postmark Driver

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

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

Next, install Guzzle and set the `driver` option in your `config/mail.php` configuration file to `postmark`. Finally, verify that your `config/services.php` configuration file contains the following options:

```
'postmark' => [
    'token' => 'your-postmark-token',
```

```
],
```

SES Driver

To use the Amazon SES driver you must first install the Amazon AWS SDK for PHP. You may install this library by adding the following line to your `composer.json` file's `require` section and running the `composer update` command:

```
"aws/aws-sdk-php": "~3.0"
```

Next, set the `driver` 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' => 'your-ses-key',
    'secret' => 'your-ses-secret',
    'region' => 'ses-region',  // e.g. us-east-1
],
```

If you need to include additional options when executing the SES `SendRawEmail` request, you may define an `options` array within your `ses` configuration:

```
'ses' => [
    'key' => 'your-ses-key',
    'secret' => 'your-ses-secret',
    'region' => 'ses-region',  // e.g. us-east-1
    'options' => [
        'ConfigurationSetName' => 'MyConfigurationSet',
        'Tags' => [
            [
                'Name' => 'foo',
                'Value' => 'bar',
            ],
        ],
    ],
],
```

Generating Mailables

In Laravel, 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` command:

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

Writing Mailables

All of a mailable class' configuration is done in the `build` method. Within this method, you may call various methods such as `from`, `subject`, `view`, and `attach` to configure the email's presentation and delivery.

Configuring The Sender

Using The `from` Method

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 use the `from` method within your mailable class' `build` method:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->from('example@example.com')
                ->view('emails.orders.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 call the `from` method in 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' => 'example@example.com', 'name' => 'App Name'],
```

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' `build` method, you may use the `view` method to specify 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:

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

You may wish to create a `resources/views/emails` 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 use the `text` method. Like the `view` method, the `text` method accepts 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:

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

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' constructor and set that data to public properties defined on the class:

```
<?php

namespace App\Mail;

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

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

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

    /**
     * Create a new message instance.
     *
```

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

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('emails.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:

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

Via The `with` Method:

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 `with` method. Typically, you will still pass data via the mailable class' constructor; however, you should set this data to `protected` or `private` properties so the data is not automatically made available to the template. Then, when calling the `with` method, pass an array of data that you wish to make available to the template:

```
<?php

namespace App\Mail;

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

class OrderShipped extends Mailable
{
    use Queueable, SerializesModels;

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

    /**
     * Create a new message instance.
     *
     * @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,
                    ]);
    }
}
```

Once the data has been passed to the `with` method, 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, use the `attach` method within the mailable class' `build` method. The `attach` method accepts the full path to the file as its first argument:

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

When attaching files to a message, you may also specify the display name and / or MIME type by passing an `array` as the second argument to the `attach` method:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
                ->attach('/path/to/file', [
                    'as' => 'name.pdf',
                    'mime' => '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 `attachFromStorage` method:

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

If necessary, you may specify the file's attachment name and additional options using the second and third arguments to the `attachFromStorage` method:

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

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

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

Raw Data Attachments

The `attachData` 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 `attachData` method accepts the raw data bytes as its first argument, the name of the file as its second argument, and an array of options as its third argument:

```
/**
 * Build the message.
 *
 * @return $this
 */
public function build()
{
    return $this->view('emails.orders.shipped')
                ->attachData($this->pdf, 'name.pdf', [
                    'mime' => '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 and retrieving the appropriate CID. 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>
```

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

Embedding Raw Data Attachments

If you already have a raw data string you wish to embed into an email template, you may use the `embedData` method on the `$message` variable:

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

    <img src="{{ $message->embedData($data, $name) }}">
</body>
```

Customizing The SwiftMailer Message

The `withSwiftMessage` method of the `Mailable` base class allows you to register a callback which will be invoked with the raw SwiftMailer message instance before sending the message. This gives you an opportunity to customize the message before it is delivered:

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

    $this->withSwiftMessage(function ($message) {
        $message->getHeaders()
                ->addTextHeader('Custom-Header', 'HeaderValue');
    });
}
```

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:

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

Then, when configuring the mailable within its `build` method, call the `markdown` method instead of the `view` method. The `markdown` method accepts the name of the Markdown template and an optional array of data to make available to the template:

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

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-crafted components:

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

Your order has been shipped!

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

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

Do not use excess indentation when writing Markdown emails. 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:

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

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:

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

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:

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

```
| Col 3 is      | Right-Aligned | $20      |
@endcomponent
```

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 in-lined 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 the `mail` 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 setting the email 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

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use App\Mail\OrderShipped;
use App\Order;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Mail;

class OrderController extends Controller
{
    /**
     * Ship the given order.
     *
     * @param  Request  $request
     * @param  int  $orderId
     * @return Response
     */
    public function ship(Request $request, $orderId)
    {
        $order = Order::findOrFail($orderId);

        // Ship order...

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

You are not limited to just specifying the "to" recipients when sending a message. You are free to set "to", "cc", and "bcc" recipients all within a single, chained method call:

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

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 contents of the mailable as a string:

```
$invoice = App\Invoice::find(1);

return (new App\Mail\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\Invoice::find(1);

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

Queueing Mail

Queueing A Mail Message

Since sending email messages can drastically lengthen 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:

```
$when = now()->addMinutes(10);

Mail::to($request->user())
    ->cc($moreUsers)
    ->bcc($evenMoreUsers)
    ->later($when, 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
{
```

```
    //
}
```

Localizing Mailables

Laravel allows you to send mailables in a locale other than the current language, 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 is being formatted and then revert back to the previous locale when formatting 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;

class User extends Model implements HasLocalePreference
{
    /**
     * Get the user's preferred locale.
     *
     * @return string
     */
    public function preferredLocale()
    {
        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:

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

Mail & 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. For more information on configuring your application per environment, check out the configuration documentation.

Universal To

Another solution provided by Laravel is to set a universal recipient of all emails sent by the framework. This way, all the emails generated by your application will be sent to a specific address, instead of the address actually specified when sending the message. This can be done via the `to` option in your `config/mail.php` configuration file:

```
'to' => [
    'address' => 'example@example.com',
    'name' => 'Example'
],
```

Mailtrap

Finally, you may use a service like 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.

Events

Laravel fires two events during the process of sending mail messages. The `MessageSending` event is fired prior to a message being sent, while the `MessageSent`

event is fired after a message has been sent. Remember, these events are fired when the mail is being *sent*, not when it is queued. You may register an event listener for this event in your `EventServiceProvider`:

```
/**
 * 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',
    ],
];
```

Laravel

Highlights

- Release Notes
- Getting Started
- Routing
- Blade Templates
- Authentication
- Authorization
- Artisan Console
- Database
- Eloquent ORM
- Testing

Resources

- Laracasts
- Laravel News
- Laracon
- Laracon EU
- Laracon AU
- Jobs
- Certification
- Forums

Partners

- Vehikl
- Tighten Co.
- Kirschbaum
- Byte 5
- 64Robots
- Cubet
- DevSquad
- Ideil
- Cyber-Duck
- ABOUT YOU
- Become A Partner

Ecosystem

- Vapor
- Forge
- Envoyer
- Horizon
- Lumen
- Nova
- Echo
- Valet
- Mix
- Spark
- Cashier
- Homestead
- Dusk
- Passport
- Scout
- Socialite

Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable and creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in most web projects.