

# Mailer and Webhook with Mailtrap



# Chapter 1: Installing the Mailer

Hey friends! Welcome to "Symfony Mailer with Mailtrap"! I'm Kevin, and I'll be your *postmaster* for this course, which is all about sending beautiful emails with Symfony's Mailer component, including adding HTML, CSS - and configuring for production. On that note, there are many services you can use on production to actually send your emails. This course will focus on one called Mailtrap: (1) because it's great and (2) because it offers a fantastic way to preview your emails. But don't worry, the concepts we'll cover are universal and can be applied to any email service. And bonus! We'll also cover how to track email *events* like bounces, opens, and link clicks by leveraging some relatively new Symfony components: Webhook and RemoteEvent.

## Transactional vs Bulk Emails

Before we start spamming, ahem, delivering important info via email, we need to clarify something: Symfony Mailer is for what's called *transactional* emails *only*. These are user-specific emails that occur when something specific happens in your app. Things like: a welcome email after a user signs up, an order confirmation email when they place an order, or even emails like a "your post was upvoted" are all examples of *transactional* emails. Symfony Mailer is *not* for bulk or marketing emails. Because of this, we don't need to worry about any kind of *unsubscribe* functionality. There are specific services for sending bulk emails or newsletters, Mailtrap can even do this via their site.

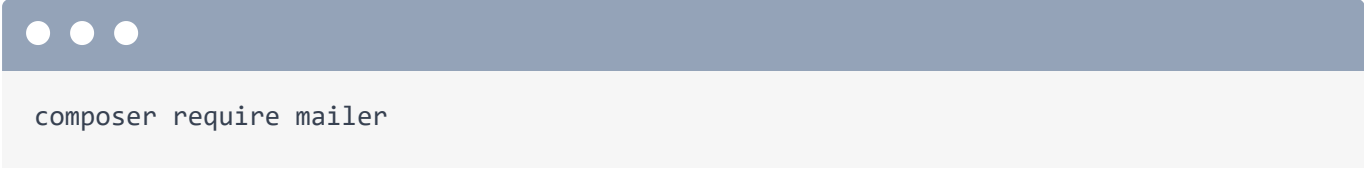
## Our Project

As always, to deliver the most bang for your screencast buck, you should totally code along with me! Download the course code on this page. When you unzip the file, you'll find a `start/` directory with the code we'll start with. Follow the `README.md` file to get the app running. I've already done this and ran `symfony serve -d` to start the web server.

Welcome to "Universal Travel": a travel agency where users can book trips to different galactic locations. Here are the currently available trips. Users *can* already book these, but there are no confirmation emails sent when they do. We're going to fix that! If I'm spending thousands of credits on a trip to Naboo, I want to know that my reservation was successful!

# Installing the Mailer Component

Step 1: let's install the Symfony Mailer! Open your terminal and run:



```
composer require mailer
```

The Symfony Flex recipe for mailer is asking us to install some Docker configuration. This is for a local SMTP server to help with previewing emails. We're going to use Mailtrap for this so say "no". Installed! Run:



```
git status
```

to see what we got. Looks like the recipe added some environment variables in `.env` and added the mailer configuration in `config/packages/mailer.yaml`.

## MAILER DSN

In your IDE, open `.env`. The Mailer recipe added this `MAILER_DSN` environment variable. This is a special URL-looking string that configures your *mailer transport*: how your emails are actually sent, like via SMTP, Mailtrap, etc. The recipe defaults to `null://null` and is perfect for local development and testing. This transport does nothing when an email is sent! It *pretends* to deliver the email, but really sends it out an airlock. We'll preview our emails in a different way.

Ok! We're ready to send our first email! Let's do that next!

## Chapter 2: Sending our First Email

Let's take a trip! "Visit Krypton", Hopefully it hasn't been destroyed yet! Without bothering to check, let's book it! I'll use name: "Kevin", email: "kevin@example.com" and just any date in the future. Hit "Book Trip".

This is the "booking details" page. Note the URL: it has a unique token specific to this booking. If a user needs to come back here later, currently, they need to bookmark this page or Slack themselves the URL if they're like me. Lame! Let's send them a confirmation email that includes a link to this page.

I want this to happen after the booking is first saved. Open `TripController` and find the `show()` method. This makes the booking: if the form is valid, create or fetch a customer and create a booking for this customer and trip. Then we redirect to the booking details page. Delightfully boring so far, just how I like my code, and weekends.

### Inject MailerInterface

I want to send an email after the booking is created. Give yourself some room by moving each method argument to its own line. Then, add `MailerInterface $mailer` to get the main service for sending emails:

```
src/Controller/TripController.php
↕ // ... lines 1 - 17
18 final class TripController extends AbstractController
19 {
↕ // ... lines 20 - 27
28     #[Route('/trip/{slug:trip}', name: 'trip_show')]
29     public function show(
↕ // ... lines 30 - 33
34         MailerInterface $mailer,
35     ): Response {
↕ // ... lines 36 - 54
55     }
56 }
```

## Create the Email

After `flush()`, which inserts the booking into the database, create a new email object:

`$email = new Email()` (the one from `Symfony\Component\Mime`). Wrap it in parentheses so we can chain methods. So what does every email need? A `from` email address: `->from()` how about `info@universal-travel.com`. A `to` email address: `->to($customer->getEmail())`. Now, the `subject`: `->subject('Booking Confirmation')`. And finally, the email needs a body: `->text('Your booking has been confirmed')` - good enough for now:

```
src/Controller/TripController.php
↕ // ... Lines 1 - 18
19 final class TripController extends AbstractController
20 {
↕ // ... Lines 21 - 29
30     public function show(
↕ // ... Lines 31 - 35
36         ): Response {
↕ // ... Lines 37 - 38
39         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... Lines 40 - 48
49             $email = (new Email())
50                 ->from('info@universal-travel.com')
51                 ->to($customer->getEmail())
52                 ->subject('Booking Confirmation')
53                 ->text('Your booking has been confirmed!')
54             ;
↕ // ... Lines 55 - 56
57         }
↕ // ... Lines 58 - 62
63     }
64 }
```

## Send the Email

Finish with `$mailer->send($email)`:

```
src/Controller/TripController.php
```

```
↕ // ... Lines 1 - 18
19 final class TripController extends AbstractController
20 {
↕ // ... Lines 21 - 29
30     public function show(
↕ // ... Lines 31 - 35
36         ): Response {
↕ // ... Lines 37 - 38
39         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... Lines 40 - 55
56             $mailer->send($email);
↕ // ... Lines 57 - 58
59         }
↕ // ... Lines 60 - 64
65     }
66 }
```

Let's test this out!

Back in our app, go back to the homepage and choose a trip. For the name, use "Steve", email, "steve@minecraft.com", any date in the future, and book the trip.

Ok... this page looks exactly the same as before. Was an email sent? Nothing in the web debug toolbar seems to indicate this...

The email was *actually* sent on the previous request - the form submit. That controller then redirected us to this page. But the web debug toolbar gives us a shortcut to access the profiler for the previous request: hover over `200` and click the profiler link to get there.

Check out the sidebar - we have a new "Emails" tab! And it shows 1 email was sent. We did it! Click it, and here's our email! The from, to, subject, and body are all what we expect.

Remember, we're using the `null` mailer transport, so this email wasn't actually sent, but it's super cool we can still preview it in the profiler!

Though ... I think we both know this email... is... pretty crappy. It doesn't give any of the useful info! No URL to the booking details page, no destination, no date, no nothing! It's so useless, I'm glad the `null` transport is just throwing it out the space window.

Let's fix that next!

## Chapter 3: Better Email

I think you, me, anyone that's ever received an email, can agree that our first email stinks. It doesn't provide any value. Let's improve it!

First, we can add a name to the email. This will show up in most email clients instead of just the email address: it just looks smoother. Wrap the `from` with `new Address()`, the one from `Symfony\Component\Mime`. The first argument is the email, and the second is the name - how about `Universal Travel`:

```
src/Controller/TripController.php
↕ // ... Lines 1 - 19
20 final class TripController extends AbstractController
21 {
↕ // ... Lines 22 - 30
31     public function show(
↕ // ... Lines 32 - 36
37         ): Response {
↕ // ... Lines 38 - 39
40         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... Lines 41 - 49
50             $email = (new Email())
51                 ->from(new Address('info@universal-travel.com', 'Universal
Travel'))
↕ // ... Lines 52 - 54
55             ;
↕ // ... Lines 56 - 59
60         }
↕ // ... Lines 61 - 65
66     }
67 }
```

We can also wrap the `to` with `new Address()` and pass `$customer->getName()` for the name:

src/Controller/TripController.php

```
↕ // ... lines 1 - 19
20 final class TripController extends AbstractController
21 {
↕ // ... lines 22 - 30
31     public function show(
↕ // ... lines 32 - 36
37     ): Response {
↕ // ... lines 38 - 39
40         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 41 - 49
50             $email = (new Email())
↕ // ... line 51
52                 ->to(new Address($customer->getEmail()))
↕ // ... lines 53 - 54
55             ;
↕ // ... lines 56 - 59
60         }
↕ // ... lines 61 - 65
66     }
67 }
```

For the `subject`, add the trip name: `'Booking Confirmation for ' . $trip->getName():`

src/Controller/TripController.php

```
↕ // ... lines 1 - 19
20 final class TripController extends AbstractController
21 {
↕ // ... lines 22 - 30
31     public function show(
↕ // ... lines 32 - 36
37     ): Response {
↕ // ... lines 38 - 39
40         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 41 - 49
50             $email = (new Email())
↕ // ... lines 51 - 52
53                 ->subject('Booking Confirmation for ' . $trip->getName())
↕ // ... line 54
55             ;
↕ // ... lines 56 - 59
60         }
↕ // ... lines 61 - 65
66     }
67 }
```



For the `text` body. We could inline all the text right here. That would get ugly, so let's use Twig! We need a template. In `templates/`, add a new `email/` directory and inside, create a new file: `booking_confirmation.txt.twig`. Twig can be used for any text format, not just `html`. A good practice is to include the format - `.html` or `.txt` - in the filename. But Twig doesn't care about that - it's just to satisfy our human brains. We'll return to this file in a second.

Back in `TripController::show()`, instead of `new Email()`, use `new TemplatedEmail()` (the one from `Symfony\Bridge\Twig`):

```
src/Controller/TripController.php
↕ // ... lines 1 - 19
20 final class TripController extends AbstractController
21 {
↕ // ... lines 22 - 30
31     public function show(
↕ // ... lines 32 - 36
37     ): Response {
↕ // ... lines 38 - 39
40         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 41 - 49
50             $email = (new TemplatedEmail())
↕ // ... lines 51 - 64
65         }
↕ // ... lines 66 - 70
71     }
72 }
```

Replace `->text()` with `->textTemplate('email/booking_confirmation.txt.twig')`:

src/Controller/TripController.php

```
↕ // ... Lines 1 - 19
20 final class TripController extends AbstractController
21 {
↕ // ... Lines 22 - 30
31     public function show(
↕ // ... Lines 32 - 36
37     ): Response {
↕ // ... Lines 38 - 39
40         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... Lines 41 - 49
50             $email = (new TemplatedEmail())
↕ // ... Lines 51 - 53
54                 ->textTemplate('email/booking_confirmation.txt.twig')
↕ // ... Lines 55 - 59
60             ;
↕ // ... Lines 61 - 64
65         }
↕ // ... Lines 66 - 70
71     }
72 }
```

To pass variables to the template, use `->context()` with

```
'customer' => $customer, 'trip' => $trip, 'booking' => $booking:
```

```
src/Controller/TripController.php
```

```
↕ // ... Lines 1 - 19
20 final class TripController extends AbstractController
21 {
↕ // ... Lines 22 - 30
31     public function show(
↕ // ... Lines 32 - 36
37         ): Response {
↕ // ... Lines 38 - 39
40         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... Lines 41 - 49
50             $email = (new TemplatedEmail())
↕ // ... Lines 51 - 54
55                 ->context([
56                     'customer' => $customer,
57                     'trip' => $trip,
58                     'booking' => $booking,
59                 ])
60             ;
↕ // ... Lines 61 - 64
65         }
↕ // ... Lines 66 - 70
71     }
72 }
```

Note that we aren't technically *rendering* the Twig template here: Mailer will do that for us before it sends the email.

This is normal, boring Twig code. Let's render the user's first name using a cheap trick, the trip name, the departure date, and a link to manage the booking. We need to use absolute URLs in emails - like <https://universal-travel.com/booking> - so we'll leverage the `url()` Twig function instead of `path()`: `{{ url('booking_show', {'uid': booking.uid}) }}`. End politely with, `Regards, the Universal Travel team`:

```
templates/email/booking_confirmation.txt.twig
```

```
1 Hey {{ customer.name|split(' ')|first }},
2
3 Get ready for your trip to {{ trip.name }}!
4
5 Departure: {{ booking.date|date('Y-m-d') }}
6
7 Manage your booking: {{ url('booking_show', {uid: booking.uid}) }}
8
9 Regards,
10 The Universal Travel Team
```

Email body done! Test it out. Back in your browser, choose a trip, name: **Steve**, email: **steve@minecraft.com**, any date in the future, and book the trip. Open the profiler for the last request and click the **Emails** tab to see the email.

Much better! Notice the **From** and **To** addresses now have names. And our text content is definitely more valuable! Copy the booking URL and paste it into your browser to make sure it goes to the right place. Looks like it, nice!

Next, we'll use [Mailtrap](#)'s testing tool for a more robust email preview.

# Chapter 4: Previewing Emails with Mailtrap (Email Testing)

Previewing emails in the profiler is okay for basic emails, but soon we'll add HTML styles and images of space cats. To properly see how our emails look, we need a more robust tool. We're going to use [Mailtrap](#)'s *email testing tool*. This gives us a real SMTP server that we can connect to, but instead of delivering emails to real inboxes, they go into a fake inbox that we can check out! It's like we send an email for real, then hack that person's account to see it... but without the hassle or all that illegal stuff!

Go to <https://mailtrap.io> and sign up for a free account. Their free tier plan has some limits but is perfect for getting started. Once you're in, you'll be on their app homepage. What we're interested in right now is *email testing*, so click that. You should see something like this. If you don't have an inbox yet, add one here.

Open that shiny new inbox. Next, we need to configure our app to send emails via the Mailtrap SMTP server. This is easy! Down here, under "Code Examples", click "PHP" then "Symfony". Copy the `MAILER_DSN`.

Because this is a sensitive value, and may vary between developers, don't add it to `.env` as that's committed to git. Instead, create a new `.env.local` file at the root of your project. Paste the `MAILER_DSN` here to override the value in `.env`.

We are set up for Mailtrap testing! That was easy! Test'r out!

Back in the app, book a new trip: Name: `Steve`, Email: `steve@minecraft.com`, any date in the future, and... book! This request takes a bit longer because it's connecting to the external Mailtrap SMTP server.

Back in Mailtrap, bam! The email's already in our inbox! Click to check it out. Here's a "Text" preview and a "Raw" view. There's also a "Spam Analysis" - cool! "Tech Info" shows all the nerdy "email headers" in an easy-to-read format.

These "HTML" tabs are greyed out because we don't have an HTML version of our email... yet... Let's change that next!

## Chapter 5: HTML-powered Emails

Emails should always have a plain-text version, but they can also have an HTML version. And that's where the fun is! Time to make this email more presentable by adding HTML!

In `templates/email/`, copy `booking_confirmation.txt.twig` and name it `booking_confirmation.html.twig`. The HTML version acts a bit like a full HTML page. Wrap everything in an `<html>` tag, add an empty `<head>` and wrap the content in a `<body>`. I'll also wrap these lines in `<p>` tags to get some spacing... and a `<br>` tag after "Regards," to add a line break.

This URL can now live in a proper `<a>` tag. Give yourself some room and cut "Manage your booking". Add an `<a>` tag with the URL as the `href` attribute and paste the text inside.

Finally, we need to tell Mailer to use this HTML template. In `TripController::show()`, above `->textTemplate()`, add `->htmlTemplate()` with `email/booking_confirmation.html.twig`.

Test it out by booking a trip: `Steve`, `steve@minecraft.com`, any date in the future, book... then check Mailtrap. The email looks the same, but now we have an HTML tab!

Oh and the "HTML Check" is really neat. It gives you a gauge of what percentage of email clients support the HTML in this email. If you didn't know, email clients are a pain in the butt: it's like the 90s all over again with different browsers. This tool helps with that. Back in the HTML tab, click the link to make sure it works. It does!

So our email now has both a text and HTML version but... it's kind of a drag to maintain both. Who uses a text-only email client anyway? Probably nobody or a very low percentage of your users.

Let's try something: in `TripController::show()`, remove the `->textTemplate()` line. Our email now only has an HTML version.

Book another trip and check the email in Mailtrap. We still have a text version? It looks almost like our text template but with some extra spacing. If you send an email with just an HTML version, Symfony Mailer automatically creates a text version but strips the tags. This is a nice

fallback, but it's not perfect. See what's missing? The link! That's... kind of critical... The link is gone because it was in the `href` attribute of the anchor tag. We lost it when the tags were stripped.

So, do we need to always manually maintain a text version? Not necessarily. Here's a little trick.

Over in your terminal, run:



```
composer require league/html-to-markdown
```

This is a package that converts HTML to markdown. Wait, what? Don't we usually convert markdown to HTML? Yes, but for HTML emails, this is perfect! And guess what? There's nothing else we need to do! Symfony Mailer automatically uses this package instead of just stripping tags if available!

Book yet another trip and check the email in Mailtrap. The HTML looks the same, but check the text version. Our anchor tag has been converted to a markdown link! It's still not perfect, but at least it's there! If you need full control, you'll need that separate text template, but, I think this is good enough. Back in your IDE, delete `booking_confirmation.txt.twig`.

Next, we'll spice up this HTML with CSS!

# Chapter 6: CSS in Email

CSS in email requires... some special care. But, pffff, we're Symfony developers! Let's recklessly go forward and see what happens!

In `email/booking_confirmation.html.twig`, add a `<style>` tag in the `<head>` and add a `.text-red` class that sets the `color` to `red`. Now, add this class to the first `<p>` tag.

In our app, book another trip for our good friend Steve. He's really racking up the parsecs! Do you think he'd be interested in the platinum Universal Travel credit card?

In Mailtrap, check the email. Ok, this text is red like we expect... so what's the problem? Check the HTML Source for a hint. Hover over the first error:

*“The `style` tag is not supported in all email clients.”*

The more important problem is the `class` attribute: it's also not supported in all email clients. We can travel to space but can't use CSS classes in emails? Yup! It's a strange world.

The solution? Pretend like it's 1999 and inline all the styles. That's right, for every tag that has a `class`, we need to find all the styles applied from the class and add them as a `style` attribute. Manually, this would suuuuck... Luckily, Symfony Mailer has you covered!

At the top of this file, add a Twig `apply` tag with the `inline_css` filter. If you're unfamiliar, the `apply` tag allows you to apply any Twig filter to a block of content. At the end of the file, write `endapply`.

Book another trip for Steve. Oops, an error! The `inline_css` filter is part of a package we don't have installed but the error message gives us the `composer require` command to install it! Copy that, jump over to your terminal and paste:

```
composer require twig/cssinliner-extra
```

Back in the app, rebook Steve's trip and check the email in Mailtrap.



The HTML looks the same but check the HTML Source. This `style` attribute was automatically added to the `<p>` tag! That's amazing and way better than doing it manually.

If your app sends multiple emails, you'll want them to have a consistent style from a real CSS file, instead of defining everything in a `<style>` tag in each template. Unfortunately, it's not as simple as linking to a CSS file in the `<head>`. That's something else that email clients don't like.

No problem!

Create a new `email.css` file in `assets/styles/`. Copy the CSS from the email template and paste it here. Back in the template, celebrate by removing the `<style>` tag.

So how can we get our email to use the external CSS file? With trickery of course!

Open `config/packages/twig.yaml` and create a `paths` key. Inside, add `%kernel.project_dir%/assets/styles: styles`. I know, this looks weird, but it creates a custom Twig namespace. Thanks to this we can now render templates inside this directory with the `@styles/` prefix. But wait a minute! `email.css` file isn't a twig template that we want to render! That's ok, we just need to access it, not parse it as Twig.

Back in `booking_confirmation.html.twig`, for `inline_css`'s argument, use `source('@styles/email.css')`. The `source()` function grabs the raw content of a file.

Jump to our app, book another trip and check the email in Mailtrap. Looks the same! The text here is red. If we check the HTML Source, the classes are no longer in the `<head>` but the styles *are* still inlined: they're being loaded from our external style sheet, it's brilliant!

Up next, let's improve the HTML and CSS to make this email worthy of Steve's inbox and the expensive trip he just booked.

# Chapter 7: Real Email Styling with Inky & Foundation CSS

To get this email looking really sharp, we need to improve the HTML and CSS.

Let's start with CSS. With standard website CSS, you've likely used a CSS framework like Tailwind (which our app uses), Bootstrap, or Foundation. Does something like this exist for emails? Yes! And it's even more important to use one for emails because there are so many email clients that render differently.

For emails, we recommend using Foundation as it has a specific framework for emails. Google "Foundation CSS" and you should find this page. Foundation

Download the starter kit for the "CSS Version". This zip file includes a `foundation-emails.css` file that's the actual "framework".

I already included this in the `tutorials/` directory. Copy it to `assets/styles/`.

In our `booking_confirmation.html.twig`, the `inline_css` filter can take multiple arguments. Make the first argument `source('@styles/foundation-emails.css')` and use `email.css` for the second argument. This will contain custom styles and overrides.

I'll open `email.css` and paste in some custom CSS for our email.

Now we need to improve our HTML. But weird news! Most of the things we use for styling websites don't work in emails. For example, we can't use Flexbox or Grid. Instead, we need to use tables for layout. Tables! Tables, inside tables, inside tables. Gross!

Luckily, there's a templating language we can use to make this easier. Search for "inky templating language" to find this page. Inky is developed by this Zurb Foundation. Zurb, Inky, Foundation... these names fit in perfectly with our space theme! And they all work together!

You can get an idea of how it works on the overview. This is the HTML needed for a simple email. It's table-hell! Click the "Switch to Inky" tab. Wow! This is much cleaner! We write in a more readable format and Inky converts it to the table-hell needed for emails.

There are even "inky components": buttons, callouts, grids, etc.

In your terminal, install an Inky Twig filter that will convert our Inky markup to HTML.

```
composer require twig/inky-extra
```

In `booking_confirmation.html.twig`, add the `inky_to_html` filter to `apply`, piping `inline_css` after. First, we apply the Inky filter, then inline the CSS.

I'll copy in some inky markup for our email. We have a `<container>`, with `<rows>` and `<columns>`. This will be a single column email, but you can have as many columns as you need. This `<spacer>` adds vertical space for breathing room.

Let's see this email in action! Book a new trip for Steve, oops, must be a date in the future, and book!

Check Mailtrap and find the email. Wow! This looks much better! We can use this little widget Mailtrap provides to see how it'll look on mobile and tablets.

Looking at the "HTML Check", seems like we have some issues, but, I think as long we're using Foundation and Inky as intended, we should be good.

Check the buttons. "Manage Booking", yep, that works. "My Account", yep, that works too. That was a lot of quick success thanks to Foundation and Inky!

Next, let's improve our email further by embedding the trip image and making the lawyers happy by adding a "terms of service" PDF attachment.

## Chapter 8: Attachments and Images

Can we add an attachment to our email? Of course! Doing this manually is a complex and delicate process. Luckily, the Symfony Mailer makes it a cinch.

In the `tutorial/` directory, you'll see a `terms-of-service.pdf` file. Move this into `assets/`, though it could live anywhere.

In `TripController::show()`, we need to get the path to this file. Add a new `string $termsPath` argument and with the `#[Autowire]` attribute and `%kernel.project_dir%/assets/terms-of-service.pdf'`.

Cool, right?

Down where we create the email, write `->attach` and see what your IDE suggests. There are two methods: `attach()` and `attachFromPath()`. `attach()` is for adding the raw content of a file (as a string or stream). Since our attachment is a real file on our filesystem, use `attachFromPath()` and pass `$termsPath` then a friendly name like `Terms of Service.pdf`. This will be the name of the file when it's downloaded. as the second. If the second argument *isn't* passed, it defaults to the file's name.

Attachment done. That was easy!

Next, let's add the trip image to the booking confirmation email. But we don't want it as an attachment. We want it embedded in the HTML. There are two ways to do this: First, the standard web way: use an `<img>` tag with an absolute URL to the image hosted on your site. But, we're going to be clever and embed the image directly into the email. This is *like* an attachment, but isn't available for download. Instead, you reference it in the HTML of your email.

First, like we did with our external CSS files, we need to make our images available in Twig. `public/imgs/` contains our trip images and they're all named as `<trip-slug.png>`.

In `config/packages/twig.yaml`, add another `paths` entry:

```
%kernel.project_dir%/public/imgs: images
```

Now we can access this directory in Twig with `@images/`. Close this file.

## The email Variable

When you use Twig to render your emails, of course you have access to the variables passed to `->context()` but there's also a secret variable available called `email`. This is an instance of `WrappedTemplatedEmail` and it gives you access to email-related things like the subject, return path, from, to, etc. The thing we're interested in is this `image()` method. This is what handles embedding images!

Let's use it!

In `booking_confirmation.html.twig`, below this `<h1>`, add an `<img>` tag with some classes: `trip-image` from our custom CSS file and `float-center` from Foundation.

For the `src`, write `{{ email.image() }}`, this is the method on that `WrappedTemplatedEmail` object. Inside, write `'@images/%s.png' | format(trip.slug)`. Add an `alt="{{ trip.name }}"` and close the tag.

Image embedded! Let's check it!

Back in the app, book a trip... and check Mailtrap. Here's our email and... here's our image! We rock! It fits perfectly and even has some nice rounded corners.

Up here, in the top right, we see "Attachment (1)" - just like we expect. Click this and choose "Terms of Service.pdf" to download it. Open it up and... there's our PDF! Our space lawyers actually made this document fun - and it only cost us \$500/hour! Investor money well spent!

Next, we're going to remove the need to manually set a `from` to each email by using events to add it globally.

*With <3 from SymphonyCasts*