

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.

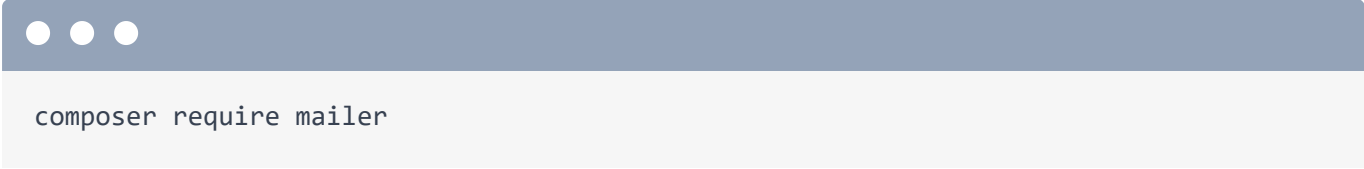
Our Project

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.

Email in the Profiler

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!

Address Object

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 the that - it's just to satisfy our human brains. We'll return to this file in a second.

Twig Email Template

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!

Fake Inbox

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`.

MAILER_DSN for Fake Inbox

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.

Email in Mailtrap

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!

HTML Email Template

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 `
` 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.

```
templates/email/booking_confirmation.html.twig
```

```
1 <html>
2 <head></head>
3 <body>
4 <p>Hey {{ customer.name|split(' ')|first }},</p>
5
6 <p>Get ready for your trip to {{ trip.name }}!</p>
7
8 <p>Departure: {{ booking.date|date('Y-m-d') }}</p>
9
10 <p>
11     <a href="{{ url('booking_show', {uid: booking.uid}) }}">
12         Manage your booking
13     </a>
14 </p>
15
16 <p>
17     Regards,<br>
18     The Universal Travel Team
19 </p>
20 </body>
21 </html>
```


Finally, we need to tell Mailer to use this HTML template. In `TripController::show()`, above `->textTemplate()`, add `->htmlTemplate()` with `email/booking_confirmation.html.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 - 49
50         $email = (new TemplatedEmail())
↕ // ... lines 51 - 53
54             ->htmlTemplate('email/booking_confirmation.html.twig')
55             ->textTemplate('email/booking_confirmation.txt.twig')
↕ // ... lines 56 - 60
61         ;
↕ // ... lines 62 - 65
66     }
↕ // ... lines 67 - 71
72 }
73 }
```

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.

Automatically Generating Text Version

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.

HTML to Markdown

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!

Add a CSS Class

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`:

```
templates/email/booking_confirmation.html.twig
1 <html>
2 <head>
3     <style>
4         .text-red {
5             color: red;
6         }
7     </style>
8 </head>
↕ // ... lines 9 - 26
27 </html>
```

Now, add this class to the first `<p>` tag:

```
templates/email/booking_confirmation.html.twig
↕ // ... lines 1 - 8
9 <body>
10 <p class="text-red">Hey {{ customer.name|split(' ')|first }},</p>
↕ // ... lines 11 - 25
26 </body>
↕ // ... lines 27 - 28
```

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.

Inline CSS

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!

inline_css Twig Filter

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`:

```
templates/email/booking_confirmation.html.twig
1  {% apply inline_css %}
2  <html>
  ⬆ // ... lines 3 - 27
28 </html>
29 {% 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!

External CSS File

Create a new `email.css` file in `assets/styles/`. Copy the CSS from the email template and paste it here:

```
assets/styles/email.css
```

```
1 .text-red {
2     color: red;
3 }
```

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!

Twig "styles" Namespace

Open `config/packages/twig.yaml` and create a `paths` key. Inside, add `%kernel.project_dir%/assets/styles: styles:`

```
config/packages/twig.yaml
```

```
1 twig:
2 // ... line 2
3     paths:
4         '%kernel.project_dir%/assets/styles': styles
5 // ... lines 5 - 9
```

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.

`inline css()` with `source()`

Back in `booking_confirmation.html.twig`, for `inline_css`'s argument, use `source('@styles/email.css')`:

```
templates/email/booking_confirmation.html.twig
1  {% apply inline_css(source('@styles/email.css')) %}
↕  // ... lines 2 - 24
```

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.

Foundation CSS for Emails

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

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:

```
templates/email/booking_confirmation.html.twig
1  {% apply inline_css(source('@styles/foundation-emails.css'),
    source('@styles/email.css')) %}
↕  // ... Lines 2 - 24
```

This will contain custom styles and overrides.

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

```
assets/styles/email.css
```

```
1 .trip-name {
2     font-size: 32px;
3 }
4
5 .accent-title {
6     color: #666666;
7 }
8
9 .trip-image {
10     border-radius: 12px;
11 }
```

Tables!

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!

Inky Templating Language

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
```

inky to html Twig Filter

In `booking_confirmation.html.twig`, add the `inky_to_html` filter to `apply`, piping `inline_css` after:

```
templates/email/booking_confirmation.html.twig
1 {% apply inky_to_html|inline_css(source('@styles/foundation-emails.css'),
  source('@styles/email.css')) %}
↕ // ... Lines 2 - 24
```

First, we apply the Inky filter, then inline the CSS.

I'll copy in some inky markup for our email.

```
templates/email/booking_confirmation.html.twig
```

```
1  {% apply inky_to_html|inline_css(source('@styles/foundation-emails.css'),
2  source('@styles/email.css')) %}
3      <container>
4          <row>
5              <columns>
6                  <spacer size="40"></spacer>
7                  <p class="accent-title">Get Ready for your trip to</p>
8                  <h1 class="trip-name">{{ trip.name }}</h1>
9              </columns>
10         </row>
11         <row>
12             <columns>
13                 <p class="accent-title">Departure: {{ booking.date|date('Y-m-d')
14             }}</p>
15             </columns>
16         </row>
17         <row>
18             <columns>
19                 <button class="expanded rounded center" href="{{
20                 url('booking_show', {uid: booking.uid}) }}">
21                     Manage Booking
22                 </button>
23                 <button class="expanded rounded center secondary" href="{{
24                 url('bookings', {uid: customer.uid}) }}">
25                     My Account
26                 </button>
27             </columns>
28         </row>
29         <row>
30             <columns>
31                 <p>We can't wait to see you there,</p>
32                 <p>Your friends at Universal Travel</p>
33             </columns>
34         </row>
35     </container>
36 {% endapply %}
```

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'`:

```
src/Controller/TripController.php
↕ // ... Lines 1 - 20
21 final class TripController extends AbstractController
22 {
↕ // ... Lines 23 - 31
32     public function show(
↕ // ... Lines 33 - 38
39         #[Autowire('%kernel.project_dir%/assets/terms-of-service.pdf')]
40         string $termsPath,
41     ): Response {
↕ // ... Lines 42 - 75
76     }
77 }
```

Cool, right?

Attachments

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`:

```
src/Controller/TripController.php
↕ // ... Lines 1 - 20
21 final class TripController extends AbstractController
22 {
↕ // ... Lines 23 - 31
32     public function show(
↕ // ... Lines 33 - 40
41     ): Response {
↕ // ... Lines 42 - 53
54         $email = (new TemplatedEmail())
↕ // ... Lines 55 - 57
58             ->attachFromPath($termsPath, 'Terms of Service.pdf')
↕ // ... Lines 59 - 64
65         ;
↕ // ... Lines 66 - 69
70     }
↕ // ... Lines 71 - 75
76 }
77 }
```

This will be the name of the file when it's downloaded. If the second argument *isn't* passed, it defaults to the file's name.

Attachment done. That was easy!

Embedding Images

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 `` 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:
```

```
config/packages/twig.yaml
```

```
1 twig:
  ↕ // ... line 2
3   paths:
  ↕ // ... line 4
5       '%kernel.project_dir%/public/imgs': images
  ↕ // ... lines 6 - 10
```

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 `` 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:

```
templates/email/booking_confirmation.html.twig
```

```
1  {% apply inky_to_html|inline_css(source('@styles/foundation-emails.css'),
   source('@styles/email.css')) %}
2      <container>
3          <row>
4              <columns>
5                  // ... lines 5 - 6
6              </columns>
7                  <h1 class="trip-name">{{ trip.name }}</h1>
8                  
12              </columns>
13          </row>
14          // ... lines 14 - 34
15      </container>
16  {% endapply %}
```

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 credits/hour! Investor credits 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.

Chapter 9: Global From (and Fun) with Email Events

I bet that most, if not every email your app sends will be *from* the same email address, something clever like `ha19000@universal-travel.com` or the tried-and-true but sleepier `info@universal-travel.com`.

Because every email will have the same *from* address, there's no point to set it in every email. Instead, let's set it globally. Oddly, there isn't any tiny config option for this. But that's great for us: it gives us a chance to learn about events! Very powerful, very nerdy.

The MessageEvent

Before an email is sent, Mailer dispatches a `MessageEvent`.

To listen to this, find your terminal and run:



```
symfony console make:listener
```

Call it `GlobalFromEmailListener`. This gives us a list of events we can listen to. We want the first one: `MessageEvent`. Start typing `Symfony` and it's auto-completed for us. Hit enter.

Listener created!

To be extra cool, let's set our global *from* address as a parameter. In `config/services.yaml`, under `parameters`, add a new one: `global_from_email`.

Special Email Address String

This will be a string, but check this out: set it to `Universal Travel`, then in angle brackets, put the email: `<info@universal-travel.com>`:


```
config/services.yaml
```

```
↕ // ... Lines 1 - 5
6 parameters:
7     global_from_email: 'Universal Travel <info@universal-travel.com>'
↕ // ... Lines 8 - 26
```

When Symfony Mailer sees a string that looks like this as an email address, it'll create the proper `Address` object with both a name and email set. Sweet!

MessageEvent Listener

Open the new class `src/EventListener/GlobalFromEmailListener.php`. Add a constructor with a `private string $fromEmail` argument and an `#[Autowire]` attribute with our parameter name: `%global_from_email%`:

```
src/EventListener/GlobalFromEmailListener.php
```

```
↕ // ... Lines 1 - 8
9 final class GlobalFromEmailListener
10 {
11     public function __construct(
12         #[Autowire('%global_from_email%')]
13         private string $fromEmail,
14     ) {
15     }
↕ // ... Lines 16 - 21
22 }
```

Down here, the `#[AsEventListener]` attribute is what *marks* this method as an event listener. We can actually remove this `event` argument - it'll be inferred from the method argument's type-hint: `MessageEvent`:

```
src/EventListener/GlobalFromEmailListener.php
```

```
↕ // ... Lines 1 - 9
10 final class GlobalFromEmailListener
11 {
↕ // ... Lines 12 - 17
18     #[AsEventListener]
19     public function onMessageEvent(MessageEvent $event): void
20     {
↕ // ... Lines 21 - 31
32 }
33 }
```

Inside, first grab the message from the event: `$message = $event->getMessage()`:

```
src/EventListener/GlobalFromEmailListener.php
↕ // ... Lines 1 - 9
10 final class GlobalFromEmailListener
11 {
↕ // ... Lines 12 - 18
19     public function onMessageEvent(MessageEvent $event): void
20     {
21         $message = $event->getMessage();
↕ // ... Lines 22 - 31
32     }
33 }
```

Jump into the `getMessage()` method to see what it returns. `RawMessage` ... jump into this and look at what classes extend it. `TemplatedEmail`! Perfect!

Back in our listener, write `if (!$message instanceof TemplatedEmail)`, and inside, `return;`:

```
src/EventListener/GlobalFromEmailListener.php
↕ // ... Lines 1 - 9
10 final class GlobalFromEmailListener
11 {
↕ // ... Lines 12 - 18
19     public function onMessageEvent(MessageEvent $event): void
20     {
↕ // ... Lines 21 - 22
23         if (!$message instanceof TemplatedEmail) {
24             return;
25         }
↕ // ... Lines 26 - 31
32     }
33 }
```

This will likely never be the case, but it's good practice to double-check. Plus, it helps our IDE know that `$message` is a `TemplatedEmail` now.

It's possible that an email might still set its own `from` address. In this case, we don't want to override it. So, add a guard clause: `if ($message->getFrom())`, `return;`:

```
src/EventListener/GlobalFromEmailListener.php
```

```
↕ // ... lines 1 - 9
10 final class GlobalFromEmailListener
11 {
↕ // ... lines 12 - 18
19     public function onMessageEvent(MessageEvent $event): void
20     {
↕ // ... lines 21 - 26
27         if ($message->getFrom()) {
28             return;
29         }
↕ // ... lines 30 - 31
32     }
33 }
```

Now, we can set the global `from: $message->from($this->fromEmail)`:

```
src/EventListener/GlobalFromEmailListener.php
```

```
↕ // ... lines 1 - 9
10 final class GlobalFromEmailListener
11 {
↕ // ... lines 12 - 18
19     public function onMessageEvent(MessageEvent $event): void
20     {
↕ // ... lines 21 - 30
31         $message->from($this->fromEmail);
32     }
33 }
```

Perfect!

Back in `TripController::show()`, remove the `->from()` for the email.

Time to test this! In our app, book a trip and check Mailtrap for the email. Drumroll... the `from` is set correctly! Our listener works! I never doubted us.

Reply-To

One more detail to make this completely airtight (like most of our ships).

Imagine a contact form where the user fills their name, email, and a message. This fires off an email with these details to your support team. In their email clients, it'd be nice if, when they hit reply, it goes to the email from the form - not your "global from".

You might think that you should set the `from` address to the user's email. But that won't work, as we're not authorized to send emails on behalf of that user. More on email security soon.

Fortunately, there's a special email header called `Reply-To` for just this scenario. When building your email, set it with `->replyTo()` and pass the user's email address.

Strap in because the booster tanks are full and ready for launch! Time to send *real* emails in production! That's next.

Chapter 10: Production Sending with Mailtrap

Alrighty, it's finally time send *real* emails in production!

Mailer Transports

Mailer comes with various ways to send emails, called "transports". This `smtp` one is what we're using for our Mailtrap testing. We *could* set up our own SMTP server to send emails... but... that's complex, and you need to do a lot of things to make sure your emails don't get marked as spam. Boo.

3rd-Party Transports

I highly, highly recommend using a 3rd-party email service. These handle all these complexities for you and Mailer provides *bridges* to many of these to make setup a breeze.

Mailtrap Bridge

We're using Mailtrap for testing but Mailtrap *also* has production sending capabilities! Fantabulous! It even has an official bridge!

At your terminal, install it with:

```
composer require symfony/mailtrap-mailer
```

After this is installed, check your IDE. In `.env`, the recipe added some `MAILER_DSN` stubs. We can get the real DSN values from Mailtrap, but first, we need to do some setup.

Sending Domain

Over in Mailtrap, we need to set up a "sending domain". This configures a domain you own to allow Mailtrap to properly send emails on its behalf.

Our lawyers are still negotiating the purchase of `universal-travel.com`, so for now, I'm using a personal domain I own: `zenstruck.com`. Add your domain here.

Once added, you'll be on this "Domain Verification" page. This is super important but Mailtrap makes it easy. Just follow the instructions until you get this green checkmark. Basically, you'll need to add a bunch of specific DNS records to your domain. DKIM, which verifies emails sent from your domain, and SPF, which authorizes Mailtrap to send emails on your domain's behalf are the most important. Mailtrap provides great documentation on these if you want to dig deeper on how exactly these work. But basically, we're telling the world that Mailtrap is allowed to send emails on our behalf.

Production MAILER_DSN

Once you have the green checkmark, click "Integrations" then "Integrate" under the "Transaction Stream" section.

We can now decide between using SMTP or API. I'll use the API, but either works. And hey! This looks familiar: like with Mailtrap testing, choose PHP, then Symfony. This is the `MAILER_DSN` we need! Copy it and jump over to your editor.

This is a sensitive environment variable, so add it to `.env.local` to avoid committing it to git. Comment out the Mailtrap testing DSN and paste below. I'll remove this comment because we like to keep life tidy.

Almost ready! Remember, we can only send emails in production *from* the domain we configured. In my case, `zenstruck.com`. Open `config/services.yaml` and update the `global_from_email` to your domain.

Let's see if this works! In your app, book a trip. This time use a *real* email address. I'll set the name to `Kevin` and I'll use my personal email: `kevin@symfonycasts.com`. As much as I love you and space travel, put your own email here to avoid spamming me. Choose a date and book!

We're on the booking confirmation page, that's a good sign! Now, check your personal email. I'll go to mine and wait... refresh... here it is! If I click it, this is exactly what we expect! The image, attachment, everything is here!

Next, let's see how we can track sent emails with Mailtrap plus add tags and metadata to improve that tracking!

Chapter 11: Email Tracking with Tags and Metadata

We're now sending emails for *realsies*. Let's just double-check our links are working... All good!

Mailtrap Email Logs

Mailtrap can do more than just deliver & debug emails: we can also track emails and email *events*. Jump over to Mailtrap and click "Email API/SMTP". This dashboard shows us an overview of each email we've sent. Click "Email Logs" to see the full list. Here's our email! Click it to see the details.

Hey! This look familiar... it's similar to the Mailtrap testing interface. We can see general details, a spam analysis and more. But this is really cool: click "Event History". This shows all the *events* that happened during the *flow* of this email. We can see when it was sent, delivered, even opened by the recipient! Each event has extra details, like the IP address that opened the email. Super useful for diagnosing email issues. Mailtrap also has a link tracking feature that, if enabled, would show which links were clicked in the email.

Back on the "Email Info" tab, scroll down a bit. Notice that the "Category" is "missing". This isn't actually a problem, but a "category" is a string that helps organize the different emails your app sends. This makes searching easier and can give us interesting stats like "how many user signup emails did we send last month?".

Email Tag_(Mailtrap Category)

Symfony Mailer calls this a "tag" that you can add to emails. The Mailtrap bridge takes this tag and converts it to their "category". Let's add one!

In `TripController::show()`, after the email creation, write:

```
$email->getHeaders()->add(new TagHeader()); - use booking as the name:
```



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

```
↕ // ... Lines 1 - 21
22 final class TripController extends AbstractController
23 {
↕ // ... Lines 24 - 32
33     public function show(
↕ // ... Lines 34 - 41
42     ): Response {
↕ // ... Lines 43 - 44
45         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... Lines 46 - 66
67             $email->getHeaders()->add(new TagHeader('booking'));
↕ // ... Lines 68 - 71
72         }
↕ // ... Lines 73 - 77
78     }
79 }
```

Email Metadata (Mailtrap Custom Variables)

Mailer also has a special *metadata* header that you can add to emails. This is a free-form key-value store for adding additional data. The Mailtrap bridge converts these to what they call "custom variables".

Let's add a couple:

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

```
↕ // ... Lines 1 - 22
23 final class TripController extends AbstractController
24 {
↕ // ... Lines 25 - 33
34     public function show(
↕ // ... Lines 35 - 42
43     ): Response {
↕ // ... Lines 44 - 45
46         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... Lines 47 - 68
69             $email->getHeaders()->add(new MetadataHeader('booking_uid', $booking-
>getUid()));
↕ // ... Lines 70 - 74
75         }
↕ // ... Lines 76 - 80
81     }
82 }
```

And:

```
src/Controller/TripController.php
↕ // ... Lines 1 - 22
23 final class TripController extends AbstractController
24 {
↕ // ... Lines 25 - 33
34     public function show(
↕ // ... Lines 35 - 42
43     ): Response {
↕ // ... Lines 44 - 45
46         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... Lines 47 - 69
70             $email->getHeaders()->add(new MetadataHeader('customer_uid',
                $customer->getUid()));
↕ // ... Lines 71 - 74
75         }
↕ // ... Lines 76 - 80
81     }
82 }
```

Attached to every *booking* email is now a customer and booking reference. Awesome!

To see how these'll look in Mailtrap, jump over to our app and book a trip (remember, we're still using *production sending* so use your personal email). Check our inbox... here it is. Back in Mailtrap, go back to the email logs... and refresh... there it is! Click it. Now, on this "Email Info" tab, we see our "booking" category! Down a bit further, here's our metadata or "custom variables".

Filtering by Category.

To filter on the "category", go to the email logs. In this search box, choose "Categories". This filter lists all the categories we've used. Select "booking" and "Search". This is already more organized than the Jeffries tubes down in engineering!

So that's production email sending with Mailtrap! To make things easier for the next chapters, let's switch back to using Mailtrap testing. In `.env.local`, uncomment the Mailtrap testing `MAILER_DSN` and comment out the production sending `MAILER_DSN`.

Next, let's use Symfony Messenger to send our emails *asynchronously*. Ooo!

Chapter 12: Async & Retryable Sending with Messenger

When we send this email, it's sent right away - *synchronously*. This means that our the user sees a delay while we connect to the mailer transport to send the email. And if there's a network issue where the email fails, the user will see a 500 error: not exactly inspiring confidence in a company that's going to strap you to a rocket.

Instead, let's send our emails *asynchronously*. This means that, during the request, the email will be sent to a queue to be processed later. Symfony Messenger is perfect for this! And we get the following benefits: faster responses for the user, automatic retries if the email fails, and the ability to flag emails for manual review if they fail too many times.

Installing Messenger & Doctrine Transport

Let's install messenger! At your terminal, run:



```
composer require messenger
```

Like Mailer, Messenger has the concept of a transport: this is where the messages are sent to be queued. We'll use the Doctrine transport as it's easiest to set up.



```
composer require symfony/doctrine-messenger
```

Back in our IDE, the recipe added this `MESSENGER_TRANSPORT_DSN` to our `.env` and it defaulted to Doctrine - perfect! This transport adds a table to our database so *technically* we should create a migration for this. But... we're going to cheat a bit and have it automatically create the table if it doesn't exist. To allow this, set `auto_setup` to `1`:

```

.env
↕ // ... Lines 1 - 40
41 ###> symfony/messenger ###
↕ // ... Lines 42 - 44
45 MESSENGER_TRANSPORT_DSN=doctrine://default?auto_setup=1
46 ###< symfony/messenger ###

```

Configuring Messenger Transports

The recipe also created this `config/packages/messenger.yaml` file. Uncomment the `failure_transport` line:

```

config/packages/messenger.yaml
1  framework:
2      messenger:
↕ // ... Line 3
4      failure_transport: failed
↕ // ... Lines 5 - 24

```

This enables the manual failure review system I mentioned earlier. Then, uncomment the `async` line under `transports`:

```

config/packages/messenger.yaml
1  framework:
2      messenger:
↕ // ... Lines 3 - 5
6      transports:
↕ // ... Line 7
8          async: '%env(MESSENGER_TRANSPORT_DSN)%'
↕ // ... Lines 9 - 24

```

This enables the transport configured with `MESSENGER_TRANSPORT_DSN` and names it `async`. It's not obvious here, but failed messages are retried 3 times, with an increasing delay between each attempt. If a message still fails after 3 attempts, it's sent to the `failure_transport`, called `failed`, so uncomment this transport too:

```
config/packages/messenger.yaml
```

```
1 framework:
2     messenger:
3     // ... lines 3 - 5
4
5     transports:
6     // ... lines 7 - 8
7
8     failed: 'doctrine://default?queue_name=failed'
9
10 // ... lines 10 - 24
```

Configuring Messenger Routing

The `routing` section is where we tell Symfony which messages should be sent to which transport. Mailer uses a specific message class for sending emails. So send `Symfony\Component\Mailer\Messenger\SendMessage` to the `async` transport:

```
config/packages/messenger.yaml
```

```
1 framework:
2     messenger:
3     // ... lines 3 - 11
4
5     routing:
6     // ... lines 13 - 14
7
8     'Symfony\Component\Mailer\Messenger\SendMessage': async
9
10 // ... lines 16 - 24
```

That's it! Symfony Messenger and Mailer dock together beautifully so there's nothing we need to change in our code.

Let's test this! Back in our app... book a trip. We're back to using Mailtrap's testing transport so we can use any email. Now watch how much faster this processes.

Boom!

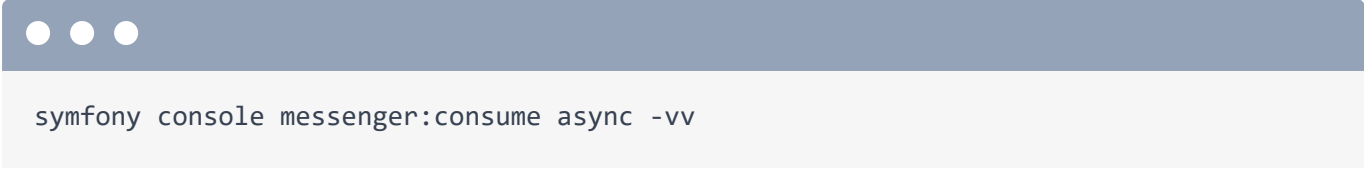
Status: Queued

Open the profiler for the last request and check out the "Emails" section. This looks normal, but notice the *Status* is "Queued". It was sent to our messenger transport, not our mailer transport. We have this new "Messages" section. Here, we can see the `SendMessage` that contains our `TemplatedEmail` object.

Jump over to Mailtrap and refresh... nothing yet. Of course! We need to process our queue.

Processing the Queue

Spin back to your terminal and run:



```
symfony console messenger:consume async -vv
```

This processes our `async` transport (the `-vv` just adds more output so we can see what's happening). Righteous! The message was received and handled successfully. Meaning: this should have *actually* sent the email.

Go check Mailtrap... it's already here! Looks correct... but... click one of our links.

What the heck? Check out the URL: that's the wrong domain! Boo. Let's find out which part of our email rocket ship has caused this and fix it next!

Chapter 13: Generating URLs in the CLI Environment

When we switched to asynchronous email sending, we broke our email links! It's using `localhost` as our domain, weird and wrong.

Back in our app, we can get a hint as to what's going on by looking at the profiler for the request that sent the email. Remember, our email is now marked as "queued". Go to the "Messages" tab and find the message: `SendMessageMessage`. Inside is the `TemplatedEmail` object. Open this up. Interesting! `htmlTemplate` is our Twig template but `html` is `null`! Shouldn't that be set to the rendered HTML from that template? This little detail is important: the email template is *not* rendered when our controller sends the message to the queue. Nope! the template isn't rendered until later, when we run `messenger:consume`.

Link Generation in the CLI

Why does this matter? Well `messenger:consume` is a CLI command, and when generating absolute URLs in the CLI, Symfony doesn't know what the domain should be (or if it should be http or https). So why does it when in a controller? In a controller, Symfony uses the current request to figure this out. In a CLI command, there is no request so it gives up and uses `http://localhost`.

Configure the Default URL

Let's just tell it what the domain should be.

Back in our IDE, open up `config/packages/routing.yaml`. Under `framework`, `routing`, these comments explain this exact issue. Uncomment `default_uri` and set it to `https://universal-travel.com` - our lawyers are close to a deal!

```
config/packages/routing.yaml
```

```
1 framework:
2     router:
3 // ... lines 3 - 4
5         default_uri: https://universal-travel.com
6 // ... lines 6 - 19
```

In development though, we need to use our local dev server's URL. For me, this is `127.0.0.1:8000` but this might be different for other team members. I know that Bob uses `bob.is.awesome:8000` and he kinda is.

Development Environment Default URL

To make this configurable, there's a trick: the Symfony CLI server sets a special environment variable with the domain called `SYMFONY_PROJECT_DEFAULT_ROUTE_URL`.

Back in our routing config, add a new section: `when@dev:`, `framework:`, `router:`, `default_uri:` and set it to `%env(SYMFONY_PROJECT_DEFAULT_ROUTE_URL)%`:

```
config/packages/routing.yaml
```

```
1 // ... lines 1 - 6
7 when@dev:
8 // ... lines 8 - 10
11     framework:
12         router:
13             default_uri: '%env(SYMFONY_PROJECT_DEFAULT_ROUTE_URL)%'
14 // ... lines 14 - 19
```

This environment variable will *only* be available if the Symfony CLI server is running *and* you're running commands via `symfony console` (not `bin/console`). To avoid an error if the variable is missing, set a default. Still under `when@dev`, add `parameters:` with `env(SYMFONY_PROJECT_DEFAULT_ROUTE_URL):` set to `http://localhost`.

```
config/packages/routing.yaml
```

```
1 // ... lines 1 - 6
7 when@dev:
8     parameters:
9         env(SYMFONY_PROJECT_DEFAULT_ROUTE_URL): 'http://localhost'
10 // ... lines 10 - 19
```

This is Symfony's standard way to set a default value for an environment variable.

Restart messenger:consume

Testing time! But first, jump back to your terminal. Because we made some changes to our config, we need to restart the `messenger:consume` command to, sort of, reload our app:

```
symfony console messenger:consume async -vv
```

Cool! The command is running again and using our sweet new Symfony config. Head back to our app... and book a trip! Quickly go back to the terminal... and we can see the message was processed.

Pop over to Mailtrap and... here it is! Moment of truth: click a link... Sweet, it's working again! Bob will be so happy!

Running messenger:consume in the Background

If you're like me, you probably find having to keep this `messenger:consume` command running in a terminal during development a drag. Plus, having to restart it every time you make a code or config change is annoying. I'm annoyed! Time to add the fun back to our functions with another Symfony CLI trick!

In your IDE, open this `.symfony.local.yaml` file. This is the Symfony CLI server config for our app. See this `workers` key? It lets us define processes to run in the background when we start the server. We already have the `tailwind` command set.

Add another worker. Call it `messenger` - though that could be anything - and set `cmd` to `['symfony', 'console', 'messenger:consume', 'async']`:

```
.symfony.local.yaml
```

```
1 workers:
```

```
↕ // ... lines 2 - 5
```

```
6 messenger:
```

```
7     cmd: ['symfony', 'console', 'messenger:consume', 'async']
```

```
↕ // ... lines 8 - 9
```

This solves the issue of needing to keep this running in a separate terminal window. But what about restarting the command when we make changes? No problemo! Add a `watch` key and

set it to `config`, `src`, `templates` and `vendor`:

```
.symfony.local.yaml
1 workers:
  ↓ // ... Lines 2 - 5
6 messenger:
  ↓ // ... Line 7
8 watch: ['config', 'src', 'templates', 'vendor']
```

If any files in these directories change, the worker will restart itself. Smart!

Back in your terminal, restart the server with `symfony server:stop` and `symfony serve -d messenger:consume` *should* be running in the background! To prove it, run:

```
symfony server:status
```

3 workers running! The actual PHP webserver, the existing `tailwind:build` worker, and our new `messenger:consume`. So cool!

Next, let's explore how to make assertions about emails in our functional tests!

Chapter 14: Emails Assertions in Functional Tests

Okay, testing time! If you've explored the codebase a bit, you may have noticed that someone (it could've been anyone... but probably a Canadian) snuck some tests into our `tests/Functional/` directory. Do these pass? Idk! Let's find out!

Jump over to your terminal and run:



```
bin/phpunit
```

Uh-oh, 1 failure. Uh-oh, because, truth time, *I'm* the friendly Canadian that added these and I know they were passing at the beginning of the course! The failure is in `BookingTest`, specifically, `testCreateBooking`:

“Expected redirect status code but got 500”

on line 38 of `BookingTest`. That's where we send the email... so if we're looking for someone to blame, I feel like we should start with the Canadian, ahem, me and my wild email-sending ways.

Foundry and Browser

Open `BookingTest.php`. If you've written functional tests with Symfony before, this may look a tad different because I'm using some rocking helper libraries. `zenstruck/foundry` gives us this `ResetDatabase` trait which wipes the database before each test. It also gives us this `Factories` trait which lets us create database fixtures in our tests. And `HasBrowser` is from another package - `zenstruck/browser` - and is essentially a user-friendly wrapper around Symfony's test client.

`testCreateBooking` is the actual test. First, we create a `Trip` in the database with these known values. Next, some pre-assertions to ensure there are no bookings or customers in the database. Now, we use `->browser()` to navigate to a trip page, fill in the booking form, and

submit. We then assert that we're redirected to a specific booking URL and check that the page contains some expected HTML. Finally, we use Foundry to make some assertions about the data in our database.

`->throwExceptions()`

Line 38 caused the failure... we're getting a 500 response code when redirecting to this booking page. 500 status codes in tests can be frustrating because it can be hard to track down the actual exception. Luckily, Browser allows us to *throw* the actual exception. At the beginning of this chain, add `->throwExceptions()`:

```
tests/Functional/BookingTest.php
↕ // ... lines 1 - 12
13 class BookingTest extends KernelTestCase
14 {
↕ // ... lines 15 - 19
20     public function testCreateBooking(): void
21     {
↕ // ... lines 22 - 30
31         $this->browser()
32             ->throwExceptions()
↕ // ... lines 33 - 42
43     ;
↕ // ... lines 44 - 52
53     }
54 }
```

Back in the terminal, run the tests again:

```
bin/phpunit
```

Now we see an exception: *Unable to find template "@images/mars.png"*. If you recall, this looks like how we're embedding the trip images into our email. It's failing because `mars.png` doesn't exist in `public/imgs`. For simplicity, let's adjust our test to use an existing image. For our fixture here, change `mars` to `iss`, and down below, for `->visit(): /trip/iss`:

```
tests/Functional/BookingTest.php
```

```
↕ // ... lines 1 - 12
13 class BookingTest extends KernelTestCase
14 {
↕ // ... lines 15 - 19
20     public function testCreateBooking(): void
21     {
22         $trip = TripFactory::createOne([
↕ // ... line 23
24             'slug' => 'iss',
↕ // ... line 25
26         ]);
↕ // ... lines 27 - 30
31         $this->browser()
↕ // ... line 32
33             ->visit('/trip/iss')
↕ // ... lines 34 - 42
43         ;
↕ // ... lines 44 - 52
53     }
54 }
```

Run the tests again!

```
bin/phpunit
```

Passing!

It *looks* like our email is being sent... but let's confirm! At the end of this test, I want to make some email assertions. Symfony *does* allow this out of the box, but I like to use a library that puts the fun back in email functional testing.

zenstruck/mailer-test

At your terminal, run:

```
composer require --dev zenstruck/mailer-test
```

Installed and configured... back in our test, enable it by adding the `InteractsWithMailer` trait:

```
tests/Functional/BookingTest.php
↕ // ... lines 1 - 13
14 class BookingTest extends KernelTestCase
15 {
16     use ResetDatabase, Factories, HasBrowser, InteractsWithMailer;
↕ // ... lines 17 - 54
55 }
```

Start simple, at the end of the test, write `$this->mailer()->assertSentEmailCount(1);`:

```
tests/Functional/BookingTest.php
↕ // ... lines 1 - 13
14 class BookingTest extends KernelTestCase
15 {
↕ // ... lines 16 - 20
21     public function testCreateBooking(): void
22     {
↕ // ... lines 23 - 54
55         $this->mailer()
56             ->assertSentEmailCount(1)
57         ;
58     }
59 }
```

Test-specific Environment Variables

Quick note: `.env.local` - where we put our *real* Mailtrap credentials - is *not* read or used in the `test` environment: our tests only load `.env` and this `.env.test` file. And in `.env`, `MAILER_DSN` is set to `null://null`. That's great! We want our tests to be fast, and not actually sending emails.

Re-run them!

```
bin/phpunit
```

`assertEmailSentTo()`

Passing - 1 email is being sent! Go back and add another assertion: `->assertEmailSentTo()`.

What email address are we expecting? The one we filled in the form:

`bruce@wayne-enterprises.com`. Copy and paste that. The second argument is the subject:

`Booking Confirmation for Visit Mars`:

```
tests/Functional/BookingTest.php
↕ // ... lines 1 - 13
14 class BookingTest extends KernelTestCase
15 {
↕ // ... lines 16 - 20
21     public function testCreateBooking(): void
22     {
↕ // ... lines 23 - 54
55         $this->mailer()
↕ // ... line 56
57         ->assertEmailSentTo('bruce@wayne-enterprises.com', 'Booking
Confirmation for Visit Mars')
58     ;
59     }
60 }
```

Run the tests!

```
bin/phpunit
```

Still passing! And notice we have 20 assertions now instead of 19.

TestEmail

But we can go further! Instead of a string for the subject in this assertion, use a closure with

`TestEmail $email` as the argument:

```
tests/Functional/BookingTest.php
```

```
↕ // ... Lines 1 - 14
15 class BookingTest extends KernelTestCase
16 {
↕ // ... Lines 17 - 21
22     public function testCreateBooking(): void
23     {
↕ // ... Lines 24 - 55
56         $this->mailer()
↕ // ... Line 57
58         ->assertEmailSentTo('bruce@wayne-enterprises.com', function(TestEmail
$email) {
↕ // ... Lines 59 - 64
65             })
66         ;
67     }
68 }
```

Inside, we can now make *loads* more assertions on this email. Since we aren't checking the subject above anymore, add this one first:

```
$email->assertSubject('Booking Confirmation for Visit Mars');
```

```
tests/Functional/BookingTest.php
```

```
↕ // ... Lines 1 - 14
15 class BookingTest extends KernelTestCase
16 {
↕ // ... Lines 17 - 21
22     public function testCreateBooking(): void
23     {
↕ // ... Lines 24 - 55
56         $this->mailer()
↕ // ... Line 57
58         ->assertEmailSentTo('bruce@wayne-enterprises.com', function(TestEmail
$email) {
59             $email
60             ->assertSubject('Booking Confirmation for Visit Mars')
↕ // ... Lines 61 - 63
64         ;
65         })
66     ;
67 }
68 }
```

And we can chain more assertions!

Write `->assert` to see what our editor suggests. Look at them all... Note the `assertTextContains` and `assertHtmlContains`. You can assert on each of these separately, but, because it's a best practice for both to contain the important details, use `assertContains()` to check both at once. Check for `Visit Mars`:

```
tests/Functional/BookingTest.php
↕ // ... lines 1 - 14
15 class BookingTest extends KernelTestCase
16 {
↕ // ... lines 17 - 21
22     public function testCreateBooking(): void
23     {
↕ // ... lines 24 - 55
56         $this->mailer()
↕ // ... line 57
58         ->assertEmailSentTo('bruce@wayne-enterprises.com', function(TestEmail
$email) {
59             $email
↕ // ... line 60
61             ->assertContains('Visit Mars')
↕ // ... lines 62 - 63
64         ;
65     })
66     ;
67 }
68 }
```

Links are important to check, so make sure the booking URL is there:

`->assertContains('/booking/')`. Now, `BookingFactory::first()->getUid()`:

tests/Functional/BookingTest.php

```
↕ // ... lines 1 - 14
15 class BookingTest extends KernelTestCase
16 {
↕ // ... lines 17 - 21
22     public function testCreateBooking(): void
23     {
↕ // ... lines 24 - 55
56         $this->mailer()
↕ // ... line 57
58         ->assertEmailSentTo('bruce@wayne-enterprises.com', function(TestEmail
$email) {
59             $email
↕ // ... lines 60 - 61
62         ->assertContains('/booking/'.BookingFactory::first()-
>getUid())
↕ // ... line 63
64         ;
65     })
66     ;
67 }
68 }
```

this fetches the first `Booking` entity in the database (which we know from above there is only the one), and gets its `uid`.

Heck! We can even check the attachment: `->assertHasFile('Terms of Service.pdf')`:

```
tests/Functional/BookingTest.php
```

```
↕ // ... lines 1 - 14
15 class BookingTest extends KernelTestCase
16 {
↕ // ... lines 17 - 21
22     public function testCreateBooking(): void
23     {
↕ // ... lines 24 - 55
56         $this->mailer()
↕ // ... line 57
58         ->assertEmailSentTo('bruce@wayne-enterprises.com', function(TestEmail
$email) {
59             $email
↕ // ... lines 60 - 62
63             ->assertHasFile('Terms of Service.pdf')
64             ;
65         })
66     ;
67 }
68 }
```

You can check the content-type and file contents via extra arguments, but I'm fine just checking that the attachment exists for now.

Go tests go!

```
bin/phpunit
```

Awesome, 25 assertions now!

[->dd\(\)](#).

One last thing: if you're ever having trouble figuring out why one of these email assertions isn't passing, chain a `->dd()`:

```
tests/Functional/BookingTest.php
```

```
↕ // ... lines 1 - 14
15 class BookingTest extends KernelTestCase
16 {
↕ // ... lines 17 - 21
22     public function testCreateBooking(): void
23     {
↕ // ... lines 24 - 55
56         $this->mailer()
↕ // ... line 57
58         ->assertEmailSentTo('bruce@wayne-enterprises.com', function(TestEmail
$email) {
59             $email
↕ // ... lines 60 - 63
64             ->dd()
65             ;
66         })
67     ;
68 }
69 }
```

and run your tests. When it hits that `dd()`, it dumps the email to help you debug. Don't forget to remove it when you're done!

Next, I want to add a second email to our app. To avoid duplication and keep things consistent, we'll create a Twig email layout that both share.

Chapter 15: Email Twig Layout

New feature time! I want to send a reminder email to customers 1 week before their booked trip. T minus 1 week to lift off people!

Symfony CLI Worker Issue

First though, we have a little problem with our Symfony CLI worker. Open `.symfony.local.yaml`. Our `messenger` worker is watching the `vendor` directory for changes. At least on some systems, there's just too many files in here to monitor and some weird things happen. No big deal: remove `vendor/`:

```
.symfony.local.yaml
1 workers:
  ⬆ // ... lines 2 - 5
6   messenger:
  ⬆ // ... line 7
8     watch: ['config', 'src', 'templates']
```

And since we changed the config, jump to your terminal and restart the webserver:

```
symfony server:stop
```

And:

```
symfony serve -d
```

Email Layout

Our new booking reminder email will have a template very similar to the booking confirmation one. To reduce duplication, and keep our snazzy emails consistent, in `templates/email/`,

create a new `layout.html.twig` template that all our emails will extend.

Copy the contents of `booking_confirmation.html.twig` and paste here. Now, remove the booking-confirmation-specific content and create an empty `content` block. I think it's fine to keep our signature here.

```
templates/email/layout.html.twig
```

```
1 {% apply inky_to_html|inline_css(source('@styles/foundation-emails.css'),
2     source('@styles/email.css')) %}
3     <container>
4         {% block content %}{% endblock %}
5     <row>
6         <columns>
7             <p>We can't wait to see you there,</p>
8             <p>Your friends at Universal Travel</p>
9         </columns>
10    </row>
11 </container>
12 {% endapply %}
```

In `booking_confirmation.html.twig`, up top here, extend this new layout and add the `content` block. Down below, copy the email-specific content and paste it inside that block. Remove everything else.

```
templates/email/booking_confirmation.html.twig
```

```
1 {% extends 'email/layout.html.twig' %}
2
3 {% block content %}
4     <row>
5         <columns>
6             <spacer size="40"></spacer>
7             <p class="accent-title">Get Ready for your trip to</p>
8             <h1 class="trip-name">{{ trip.name }}</h1>
9             
13         </columns>
14     </row>
15     <row>
16         <columns>
17             <p class="accent-title">Departure: {{ booking.date|date('Y-m-d') }}
18         </p>
19     </columns>
20 </row>
21 <row>
22     <columns>
23         <button class="expanded rounded center" href="{{ url('booking_show',
24             {uid: booking.uid}) }}">
25             Manage Booking
26         </button>
27         <button class="expanded rounded center secondary" href="{{
28             url('bookings', {uid: customer.uid}) }}">
29             My Account
30     </columns>
31 </row>
32 {% endblock %}
```

Let's make sure the booking confirmation email still works - and we have tests for that! Back in the terminal, run them with:

```
bin/phpunit
```

Green! That's a good sign. Let's be doubly sure by checking it in Mailtrap. In the app, book a trip... and check Mailtrap. It still looks fantastic!

Time to bang out the reminder email!


Booking Reminder Flag

After an email reminder is sent, we need to mark the booking so that we don't annoy the customer with multiple reminders. Let's add a new flag for this to the `Booking` entity.

In your terminal, run:

```
symfony make:entity Booking
```

Oops!



```
symfony console make:entity Booking
```

Add a new field called `reminderSentAt`, type `datetime_immutable`, nullable? Yes. This is a common pattern I use for these type of *flag* fields instead of a simple `boolean`. `null` means `false` and a date means `true`. It works the same, but gives us a bit more info.

Hit enter to exit the command.

In the `Booking` entity... here's our new property, and down here, the getter and setter.

Finding Bookings to Remind

Next, we need a way to find all bookings that need a reminder sent. Perfect job for `BookingRepository`! Add a new method called `findBookingsToRemind()`, return type: `array`. Add a docblock to show it returns an array of `Booking` objects:


```
src/Repository/BookingRepository.php
```

```
↕ // ... Lines 1 - 12
13 class BookingRepository extends ServiceEntityRepository
14 {
↕ // ... Lines 15 - 51
52     /**
53      * @return Booking[]
54      */
55     public function findBookingsToRemind(): array
56     {
↕ // ... Lines 57 - 65
66     }
67 }
```

Inside, `return $this->createQueryBuilder()`, alias `b`. Chain

```
->andWhere('b.reminderSentAt IS NULL'), ->andWhere('b.date <= :future'),
->andWhere('b.date > :now') filling in the placeholders with
->setParameter('future', new \DateTimeImmutable('+7 days')) and
->setParameter('now', new \DateTimeImmutable('now')). Finish with
->getQuery()->getResult():
```

```
src/Repository/BookingRepository.php
```

```
↕ // ... Lines 1 - 12
13 class BookingRepository extends ServiceEntityRepository
14 {
↕ // ... Lines 15 - 54
55     public function findBookingsToRemind(): array
56     {
57         return $this->createQueryBuilder('b')
58             ->andWhere('b.reminderSentAt IS NULL')
59             ->andWhere('b.date <= :future')
60             ->andWhere('b.date > :now')
61             ->setParameter('future', new \DateTimeImmutable('+7 days'))
62             ->setParameter('now', new \DateTimeImmutable('now'))
63             ->getQuery()
64             ->getResult()
65         ;
66     }
67 }
```

Pending Reminder Booking Fixture

In `AppFixtures`, down here, we create some fake bookings. Add one that will for sure trigger a reminder email to be sent: `BookingFactory::createOne()`, inside,

```
'trip' => $arrakis, 'customer' => $clark and, this is the important part,  
'date' => new \DateTimeImmutable('+6 days');
```

```
src/DataFixtures/AppFixtures.php
```

```
↕ // ... lines 1 - 10  
11 class AppFixtures extends Fixture  
12 {  
13     public function load(ObjectManager $manager): void  
14     {  
↕ // ... lines 15 - 87  
88         BookingFactory::createOne([  
89             'trip' => $arrakis,  
90             'customer' => $clark,  
91             'date' => new \DateTimeImmutable('+6 days'),  
92         ]);  
93     }  
94 }
```

Clearly between now and 7 days from now.

"Migration"

We made changes to the structure of our database. Normally, we should be creating a migration... but, we aren't using migrations. So, we'll just force update the schema. In your terminal, run:

```
symfony console doctrine:schema:update --force
```

Then, reload the fixtures:

```
symfony console doctrine:fixture:load
```

That all worked, great!

Next, we'll create a new reminder email and a CLI command to send them!

Chapter 16: Email from CLI Command

We've done the prep work for our reminder email feature. Now, let's actually create and send the emails!

Reminder Email Template

In `templates/email`, the new email template will be super similar to `booking_confirmation.html.twig`. Copy that file and name it `booking_reminder.html.twig`. Inside, I don't want to spend too much time on this, so just change the accent title to say "Coming soon!":

```
templates/email/booking_reminder.html.twig
```

```
1 {% extends 'email/layout.html.twig' %}
2
3 {% block content %}
4     <row>
5         <columns>
6             <spacer size="40"></spacer>
7             <p class="accent-title">Coming soon!</p>
8             <h1 class="trip-name">{{ trip.name }}</h1>
9             
13         </columns>
14     </row>
15     <row>
16         <columns>
17             <p class="accent-title">Departure: {{ booking.date|date('Y-m-d') }}
18         </p>
19         </columns>
20     </row>
21     <row>
22         <columns>
23             <button class="expanded rounded center" href="{{ url('booking_show',
24                 {uid: booking.uid}) }}">
25                 Manage Booking
26             </button>
27             <button class="expanded rounded center secondary" href="{{
28                 url('bookings', {uid: customer.uid}) }}">
29                 My Account
30             </button>
31         </columns>
32     </row>
33 {% endblock %}
```

Ship it! Accidental space pun!

Send Reminder Command

The logic to send the emails needs to be something we can schedule to run every hour or every day. Perfect job for a CLI command! At your terminal, run:

```
symfony make:command
```

Bah!

```
symfony console make:command
```

Call it: `app:send-booking-reminders`.

Go check it out! `src/Command/SendBookingRemindersCommand.php`. Change the description to "Send booking reminder emails":

```
src/Command/SendBookingRemindersCommand.php
↕ // ... Lines 1 - 17
18 #[AsCommand(
↕ // ... Line 19
20     description: 'Send booking reminder emails',
21 )]
22 class SendBookingRemindersCommand extends Command
↕ // ... Lines 23 - 70
```

In the constructor, autowire & set properties for `BookingRepository`, `EntityManagerInterface` and `MailerInterface`:

```
src/Command/SendBookingRemindersCommand.php
↕ // ... Lines 1 - 21
22 class SendBookingRemindersCommand extends Command
23 {
24     public function __construct(
25         private BookingRepository $bookingRepo,
26         private EntityManagerInterface $em,
27         private MailerInterface $mailer,
28     ) {
29         parent::__construct();
30     }
↕ // ... Lines 31 - 68
69 }
```

This command doesn't need any arguments or options, so remove the `configure()` method entirely.

Clear out the guts of `execute()`. Start by adding a nice:

`$io->title('Sending booking reminders')`. Then, grab the bookings that need reminders sent, with `$bookings = $this->bookingRepo->findBookingsToRemind()`.

Easy Progress Bar

To be the absolute best, let's show a progress bar as we loop over the bookings. The `$io` object has a trick for this. Write

```
foreach ($io->progressIterate($bookings) as $booking) . This handles all the boring progress bar logic for us! Inside, we need to create a new email. In TripController, copy that email - including these headers, and paste it here.
```

But we need to adjust this a bit: remove the attachment. And for the subject: replace "Confirmation" with "Reminder". Above, add some variables for convenience:

```
$customer = $booking->getCustomer() and $trip = $booking->getTrip(). Down here, keep the same metadata, but change the tag to booking_reminder. This will help us better distinguish these emails in Mailtrap.
```

Oh, and of course, change the template to `booking_reminder.html.twig`.

Still in the loop, send the email with `$this->mailer->send($email)` and mark the booking as having the reminder sent with

```
$booking->setReminderSentAt(new \DateTimeImmutable('now')).
```

Perfect! Outside the loop, call `$this->em->flush()` to save the changes to the database.

Finally, celebrate with

```
$io->success(sprintf('Sent %d booking reminders', count($bookings))).
```

```
↕ // ... Lines 1 - 21
22 class SendBookingRemindersCommand extends Command
23 {
↕ // ... Lines 24 - 31
32     protected function execute(InputInterface $input, OutputInterface $output):
    int
33     {
34         $io = new SymfonyStyle($input, $output);
35
36         $io->title('Sending booking reminders');
37
38         $bookings = $this->bookingRepo->findBookingsToRemind();
39
40         foreach ($io->progressIterate($bookings) as $booking) {
41             $trip = $booking->getTrip();
42             $customer = $booking->getCustomer();
43
44             $email = (new TemplatedEmail())
45                 ->to(new Address($customer->getEmail()))
46                 ->subject('Booking Reminder for '.$trip->getName())
47                 ->htmlTemplate('email/booking_reminder.html.twig')
48                 ->context([
49                     'customer' => $customer,
50                     'trip' => $trip,
51                     'booking' => $booking,
52                 ])
53             ;
54
55             $email->getHeaders()->add(new TagHeader('booking_reminder'));
56             $email->getHeaders()->add(new MetadataHeader('booking_uid', $booking-
>getUid()));
57             $email->getHeaders()->add(new MetadataHeader('customer_uid',
$customer->getUid()));
58
59             $this->mailer->send($email);
60             $booking->setReminderSentAt(new \DateTimeImmutable('now'));
61         }
62
63         $this->em->flush();
64
65         $io->success(sprintf('Sent %d booking reminders', count($bookings)));
66
67         return Command::SUCCESS;
68     }
69 }
```

Testing time! Pop over to your terminal. To be sure we have a booking that needs a reminder sent, reload the fixtures with:

```
symfony console doctrine:fixture:load
```

Now, run our new command!

```
symfony console app:send-booking-reminders
```

Nice, 1 reminder sent! And the output will impress our colleagues! Before we check Mailtrap, run the command again:

```
symfony console app:send-booking-reminders
```

"Sent 0 booking reminders". Perfect! Our logic to mark bookings as having reminders sent works!

Now check Mailtrap... here it is! As expected, it looks super similar to our confirmation email but, it says "Coming soon!" here: it's using the new template.

X-Tag and X-Metadata

When using "Mailtrap Testing", Mailer tags and metadata are not converted to Mailtrap categories and custom variables like they are when sent in production. But you can still make sure they're being sent! Click this "Tech Info" tab and scroll down a bit. When Mailer doesn't know how to convert tags and metadata, it adds them as these generic custom headers: `X-Tag` and `X-Metadata`.

Sure enough, `X-Tag` is `booking_reminder`. Awesome, that's what we expect too!

Ok, new feature? Check! Test for the new feature? That's next!

Chapter 17: Test for CLI Command

The captain is tired of people running after the rocket because they show up late! That's why we created a command to send reminder emails! Problem solved! Now let's write a test to ensure it *keeps working*. "New feature, new test", that's my motto!

Jump over to your terminal and run:

```
symfony console make:test
```

Type? `KernelTestCase`. Name? `SendBookingRemindersCommandTest`.

SendBookingRemindersCommandTest

In our IDE, the new class was added to `tests/`. Open it up and move the class to a new namespace: `App\Tests\Functional\Command`, to keep things organized.

Perfect. First, clear out the guts and add some behavior traits:

```
use ResetDatabase, Factories, InteractsWithMailer;
```

```
tests/Functional/Command/SendBookingRemindersCommandTest.php
```

```
↕ // ... lines 1 - 9
10 class SendBookingRemindersCommandTest extends KernelTestCase
11 {
12     use ResetDatabase, Factories, InteractsWithMailer;
↕ // ... lines 13 - 22
23 }
```

Stub out two tests: `public function testNoRemindersSent()` with `$this->markTestIncomplete()` and `public function testRemindersSent()`. Also mark it incomplete:

```
tests/Functional/Command/SendBookingRemindersCommandTest.php
```

```
↕ // ... lines 1 - 9
10 class SendBookingRemindersCommandTest extends KernelTestCase
11 {
↕ // ... lines 12 - 13
14     public function testNoRemindersSent()
15     {
16         $this->markTestIncomplete();
17     }
18
19     public function testRemindersSent()
20     {
21         $this->markTestIncomplete();
22     }
23 }
```

Back in the terminal run the tests with:

```
bin/phpunit
```

Testing TODO List

Check it out, our original two tests are passing, the two *dots*, and these *I*'s are the new incomplete tests. I love this pattern: write test stubs for a new feature, then make a game of removing the incompletes one-by-one until they're all gone. *Then*, the feature is done!

Symfony has some out-of-the-box tooling for testing commands, but I like to use a package that wraps these up into a nicer experience. Install it with:

zenstruck/console-test

```
composer require --dev zenstruck/console-test
```

To enable this package's helpers, add a new *behavior* trait to our test: `InteractsWithConsole`:

```
tests/Functional/Command/SendBookingRemindersCommandTest.php
```

```
↕ // ... Lines 1 - 10
11 class SendBookingRemindersCommandTest extends KernelTestCase
12 {
13     use ResetDatabase, Factories, InteractsWithMailer, InteractsWithConsole;
↕ // ... Lines 14 - 26
27 }
```

We're ready to knock down those I's!

testNoRemindersSent().

The first test is easy: we want to ensure that, when there's no bookings to remind, the command doesn't send any emails. Write `$this->executeConsoleCommand()` and just the command name: `app:send-booking-reminders`. Ensure the command ran successfully with `->assertSuccessful()` and `->assertOutputContains('Sent 0 booking reminders')`:

```
tests/Functional/Command/SendBookingRemindersCommandTest.php
```

```
↕ // ... Lines 1 - 10
11 class SendBookingRemindersCommandTest extends KernelTestCase
12 {
↕ // ... Lines 13 - 14
15     public function testNoRemindersSent()
16     {
17         $this->executeConsoleCommand('app:send-booking-reminders')
18             ->assertSuccessful()
19             ->assertOutputContains('Sent 0 booking reminders')
20     ;
21     }
↕ // ... Lines 22 - 26
27 }
```

testRemindersSent().

Arrange

On to the next test! This one is more involved: we need to create a booking that is eligible for a reminder. Create the booking fixture with `$booking = BookingFactory::createOne()`. Pass an array with `'trip' => TripFactory::new()`, and inside that, another array with `'name' => 'Visit Mars', 'slug' => 'iss'` (to avoid the image issue). The booking also

needs a customer: `'customer' => CustomerFactory::new()`. All we care about is the customer's email: `'email' => 'steve@minecraft.com'`. Finally, the booking date: `'date' => new \DateTimeImmutable('+4 days')`:

```
tests/Functional/Command/SendBookingRemindersCommandTest.php
↕ // ... lines 1 - 14
15 class SendBookingRemindersCommandTest extends KernelTestCase
16 {
↕ // ... lines 17 - 26
27     public function testRemindersSent()
28     {
29         $booking = BookingFactory::createOne([
30             'trip' => TripFactory::new([
31                 'name' => 'Visit Mars',
32                 'slug' => 'iss',
33             ]),
34             'customer' => CustomerFactory::new(['email' =>
'steve@minecraft.com']),
35             'date' => new \DateTimeImmutable('+4 days'),
36         ]);
↕ // ... lines 37 - 56
57     }
58 }
```

Phew! We have a booking in the database that needs a reminder sent. This test's setup, or *arrange* step, is done.

Pre-Assertion

Add a pre-assertion to ensure this booking hasn't had a reminder sent:

```
$this->assertNull($booking->getReminderSentAt());
```

```
tests/Functional/Command/SendBookingRemindersCommandTest.php
↕ // ... lines 1 - 14
15 class SendBookingRemindersCommandTest extends KernelTestCase
16 {
↕ // ... lines 17 - 26
27     public function testRemindersSent()
28     {
↕ // ... lines 29 - 37
38         $this->assertNull($booking->getReminderSentAt());
↕ // ... lines 39 - 56
57     }
58 }
```

Act

Now for the *act* step: `$this->executeConsoleCommand('app:send-booking-reminders')`
`->assertSuccessful()->assertOutputContains('Sent 1 booking reminders')`:

```
tests/Functional/Command/SendBookingRemindersCommandTest.php
↕ // ... lines 1 - 14
15 class SendBookingRemindersCommandTest extends KernelTestCase
16 {
↕ // ... lines 17 - 26
27     public function testRemindersSent()
28     {
↕ // ... lines 29 - 39
40         $this->executeConsoleCommand('app:send-booking-reminders')
41         ->assertSuccessful()
42         ->assertOutputContains('Sent 1 booking reminders')
43     ;
↕ // ... lines 44 - 56
57     }
58 }
```

Assert

Onto the *assert* phase to ensure the email was sent. In `BookingTest`, copy the email assertion and paste it here. Make a few adjustments: the email is `steve@minecraft.com`, subject is `Booking Reminder for Visit Mars` and this email doesn't have an attachment, so remove that assertion entirely:

```
tests/Functional/Command/SendBookingRemindersCommandTest.php
```

```
↕ // ... lines 1 - 14
15 class SendBookingRemindersCommandTest extends KernelTestCase
16 {
↕ // ... lines 17 - 26
27     public function testRemindersSent()
28     {
↕ // ... lines 29 - 44
45         $this->mailer()
46             ->assertSentEmailCount(1)
47             ->assertEmailSentTo('steve@minecraft.com', function(TestEmail $email)
48     {
49         $email
49             ->assertSubject('Booking Reminder for Visit Mars')
50             ->assertContains('Visit Mars')
51             ->assertContains('/booking/'.BookingFactory::first()-
52 >getUid())
52         ;
53     })
54     ;
↕ // ... lines 55 - 56
57     }
58 }
```

Finally, write an assertion that the command updated the booking in the database.

```
$this->assertNotNull($booking->getReminderSentAt());
```

```
tests/Functional/Command/SendBookingRemindersCommandTest.php
```

```
↕ // ... lines 1 - 14
15 class SendBookingRemindersCommandTest extends KernelTestCase
16 {
↕ // ... lines 17 - 26
27     public function testRemindersSent()
28     {
↕ // ... lines 29 - 55
56         $this->assertNotNull($booking->getReminderSentAt());
57     }
58 }
```

Moment of truth! Run the tests:

```
bin/phpunit
```

All green!

Outside-In Testing

I find these type of *outside-in* tests really fun and easy to write because you don't have to worry too much about testing the inner logic and they mimic how a user interacts with your app. It's no accident that the assertions are focused on what the user should see and some high level post-interaction checks, like checking something in the database.

Now that we have tests for both of our email sending paths, let's take a victory lap & refactor *with confidence* to remove duplication.

Chapter 18: Email Factory Service

Our app sends two emails: in `SendBookingRemindersCommand`, and `TripController::show()`. There is... a lot of duplication here. It hurts my eyes! But no worries! We can reorganize this into an *email factory* service. And because we have tests covering both emails, we can refactor and be confident that we haven't broken anything. I can't say it enough: I love tests!

BookingEmailFactory

Start by creating a new class: `BookingEmailFactory` in the `App\Email` namespace. Add a constructor, copy the `$termsPath` argument from `TripController::show()`, paste it here, and make it a private property:

```
src/Email/BookingEmailFactory.php
↕ // ... lines 1 - 11
12 class BookingEmailFactory
13 {
14     public function __construct(
15         #[Autowire('%kernel.project_dir%/assets/terms-of-service.pdf')]
16         private string $termsPath,
17     ) {
18     }
↕ // ... lines 19 - 54
55 }
```

Now, stub out two *factory* methods: `public function createBookingConfirmation()`, which will accept `Booking $booking`, and return `TemplatedEmail`. Then, `public function createBookingReminder(Booking $booking)` also returning a `TemplatedEmail`:


```
src/Email/BookingEmailFactory.php
```

```
↕ // ... lines 1 - 11
12 class BookingEmailFactory
13 {
↕ // ... lines 14 - 19
20     public function createBookingConfirmation(Booking $booking): TemplatedEmail
21     {
↕ // ... lines 22 - 25
26     }
↕ // ... line 27
28     public function createBookingReminder(Booking $booking): TemplatedEmail
29     {
↕ // ... lines 30 - 33
34     }
↕ // ... lines 35 - 54
55 }
```

Create a method to house that darn duplication: `private function createEmail()`, with arguments `Booking $booking` and `string $tag` that returns a `TemplatedEmail`:

```
src/Email/BookingEmailFactory.php
```

```
↕ // ... lines 1 - 11
12 class BookingEmailFactory
13 {
↕ // ... lines 14 - 35
36     private function createEmail(Booking $booking, string $tag): TemplatedEmail
37     {
↕ // ... lines 38 - 53
54     }
55 }
```

Jump to `TripController::show()`, copy *all* the email creation code, and paste it here. Up top, we need two variables: `$customer = $booking->getCustomer()` and `$trip = $booking->getTrip()`. Remove `attachFromPath()`, `subject()`, and `htmlTemplate()`. In this `TagHeader`, use the passed `$tag` variable. We can leave the metadata the same. Finally, return the `$email`:

```
src/Email/BookingEmailFactory.php
```

```
↕ // ... lines 1 - 11
12 class BookingEmailFactory
13 {
↕ // ... lines 14 - 35
36     private function createEmail(Booking $booking, string $tag): TemplatedEmail
37     {
38         $customer = $booking->getCustomer();
39         $trip = $booking->getTrip();
40         $email = (new TemplatedEmail())
41             ->to(new Address($customer->getEmail()))
42             ->context([
43             'customer' => $customer,
44             'trip' => $trip,
45             'booking' => $booking,
46         ])
47         ;
48
49         $email->getHeaders()->add(new TagHeader($tag));
50         $email->getHeaders()->add(new MetadataHeader('booking_uid', $booking-
->getUid()));
51         $email->getHeaders()->add(new MetadataHeader('customer_uid', $customer-
->getUid()));
52
53         return $email;
54     }
55 }
```

With our shared logic in place, use it in `createBookingConfirmation()`. Write `return $this->createEmail()`, passing the `$booking` variable and `booking` for the tag. Now, `->subject()`, copy this from `TripController::show()`, changing the `$trip` variable to `$booking->getTrip()`. Finally, `->htmlTemplate('email/booking_confirmation.html.twig')`:

```
src/Email/BookingEmailFactory.php
```

```
↕ // ... lines 1 - 11
12 class BookingEmailFactory
13 {
↕ // ... lines 14 - 19
20     public function createBookingConfirmation(Booking $booking): TemplatedEmail
21     {
22         return $this->createEmail($booking, 'booking')
23             ->subject('Booking Confirmation for '.$booking->getTrip()->getName())
24             ->htmlTemplate('email/booking_confirmation.html.twig')
25     ;
26 }
↕ // ... lines 27 - 54
55 }
```

For `createBookingReminder()`, copy the insides of `createBookingConfirmation()` and paste here. Change the tag to `booking_reminder`, the subject to `Booking Reminder`, and the template to `email/booking_reminder.html.twig`:

```
src/Email/BookingEmailFactory.php
```

```
↕ // ... lines 1 - 11
12 class BookingEmailFactory
13 {
↕ // ... lines 14 - 19
20     public function createBookingConfirmation(Booking $booking): TemplatedEmail
21     {
22         return $this->createEmail($booking, 'booking')
23             ->subject('Booking Confirmation for '.$booking->getTrip()->getName())
24             ->htmlTemplate('email/booking_confirmation.html.twig')
25     ;
26 }
↕ // ... lines 27 - 54
55 }
```

The Refactor

Now the fun part! *Using* our factory and *removing* a whole wack of code!

In `TripController::show()`, instead of injecting `$termsPath`, inject `BookingEmailFactory $emailFactory`:

```
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         BookingEmailFactory $emailFactory,
37     ): Response {
↕ // ... lines 38 - 58
59     }
60 }
```

Delete all the email creation code and inside `$mailer->send()`, write `$emailFactory->createBookingConfirmation($booking)`:

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

```
↕ // ... lines 1 - 18
19 final class TripController extends AbstractController
20 {
↕ // ... lines 21 - 29
30     public function show(
↕ // ... lines 31 - 36
37     ): Response {
↕ // ... lines 38 - 39
40         if ($form->isSubmitted() && $form->isValid()) {
↕ // ... lines 41 - 49
50             $mailer->send($emailFactory->createBookingConfirmation($booking));
↕ // ... lines 51 - 52
53         }
↕ // ... lines 54 - 58
59     }
60 }
```

Over in `SendBookingRemindersCommand`, again, remove all the email creation code. Up in the constructor, autowire `private BookingEmailFactory $emailFactory`:

```
src/Command/SendBookingRemindersCommand.php
```

```
↕ // ... Lines 1 - 18
19 class SendBookingRemindersCommand extends Command
20 {
21     public function __construct(
↕ // ... Lines 22 - 24
25         private BookingEmailFactory $emailFactory,
26     ) {
↕ // ... Line 27
28     }
↕ // ... Lines 29 - 48
49 }
```

Down here, inside `$this->mailer->send()`, write

`$this->emailFactory->createBookingReminder($booking)`:

```
src/Command/SendBookingRemindersCommand.php
```

```
↕ // ... Lines 1 - 18
19 class SendBookingRemindersCommand extends Command
20 {
↕ // ... Lines 21 - 29
30     protected function execute(InputInterface $input, OutputInterface $output):
    int
31     {
↕ // ... Lines 32 - 37
38         foreach ($io->progressIterate($bookings) as $booking) {
39             $this->mailer->send($this->emailFactory-
>createBookingReminder($booking));
↕ // ... Line 40
41         }
↕ // ... Lines 42 - 47
48     }
49 }
```

Test It

Oh yeah, that felt good! But did we break anything? We Canadians are known for being a bit wild. Check by running the tests:

```
bin/phpunit
```

Uh oh, a failure! Good thing we have these tests, eh?

The failure comes from `BookingTest`:

“Message does not include file with filename [Terms of Service.pdf].”

Fix It

Easy fix! During our refactor, I forgot to attach the thrilling terms of service PDF to the booking confirmation email. And our customers depend on that. Find

`BookingEmailFactory::createBookingConfirmation()`, and add

`->attachFromPath($this->termsPath, 'Terms of Service.pdf')`:

```
src/Email/BookingEmailFactory.php
↕ // ... lines 1 - 11
12 class BookingEmailFactory
13 {
↕ // ... lines 14 - 19
20     public function createBookingConfirmation(Booking $booking): TemplatedEmail
21     {
22         return $this->createEmail($booking, 'booking')
↕ // ... lines 23 - 24
25         ->attachFromPath($this->termsPath, 'Terms of Service.pdf')
26     ;
27     }
↕ // ... lines 28 - 55
56 }
```

Re-run the tests:

```
bin/phpunit
```

Passing! Successful refactor? Check!

Next, let's switch gears a bit and dive into *two* new Symfony components to consume the email webhook *events* from Mailtrap.

Chapter 19: The Webhook Component for Email Events

In Mailtrap, when we send emails in production, remember that we can check each email: was it sent, delivered, opened, bounced (which is important!) and more. Mailtrap lets us set a webhook URL so it can send info about these events *to* us.

Webhook & RemoteEvent Components

As a bonus, we get to discover *two* new Symfony components! Find your terminal and install them:

```
composer require webhook remote-event
```

The webhook component gives us a single endpoint to send all webhooks to. It parses the data sent to us - called the payload, converts it to a *remote event* object, and sends it to a *consumer*. You can think of remote events as similar to Symfony events. Instead of your app dispatching an event, a third-party service does it - hence *remote event*. And instead of *event listeners*, we say that remote events have *consumers*.

Run

```
git status
```

to see what the recipe added: `config/routes/webhook.yaml`. Cool! That adds the webhook controller. Check out the route with:

```
symfony console debug:route webhook
```

Check the first one. The path is `/webhook/{type}`. So now we need to configure some sort of a *type*.

3rd party webhooks - like from Mailtrap or a payment processor or a supernova alert system - can send us *wildly* different payloads, we typically need to create our own parsers and remote events. Since email events are pretty standard, Symfony provides some out-of-the-box remote events for these: `MailerDeliveryEvent` and `MailerEngagementEvent`. Some mailer bridges, including the Mailtrap bridge we're using, provide parsers for each service's webhook payload to create these objects. We just need to set it up.

Mailtrap Parser Configuration

In `config/packages/`, create a `webhook.yaml` file. Add: `framework`, `webhook`, `routing`, `mailtrap` (this is the *type* used in the URL), and then `service`. To figure out the Mailtrap parser service id, pop over to the [Symfony Webhook documentation](#). Find the service id for the Mailtrap parser, copy it... and paste it here:

```
config/packages/webhook.yaml
```

```
1 framework:
2     webhook:
3         routing:
4             mailtrap:
5                 service: mailer.webhook.request_parser.mailtrap
```

EmailEventConsumer

Now we need a consumer. Create a new class called `EmailEventConsumer` in the `App\Webhook` namespace. This needs to implement `ConsumerInterface` from `RemoteEvent`. Add the necessary `consume()` method. To tell Symfony which webhook *type* we want this to consume, add the `#[AsRemoteEventConsumer]` attribute with `mailtrap`:


```
src/Webhook/EmailEventConsumer.php
```

```
↕ // ... lines 1 - 10
11 #[AsRemoteEventConsumer('mailtrap')]
12 class EmailEventConsumer implements ConsumerInterface
13 {
↕ // ... lines 14 - 16
17     public function consume(RemoteEvent $event): void
18     {
↕ // ... line 19
20     }
21 }
```

Above `consume()`, add a docblock to help our IDE:

```
@param MailerDeliveryEvent|MailerEngagementEvent $event:
```

```
src/Webhook/EmailEventConsumer.php
```

```
↕ // ... lines 1 - 11
12 class EmailEventConsumer implements ConsumerInterface
13 {
14     /**
15      * @param MailerDeliveryEvent|MailerEngagementEvent $event
16      */
17     public function consume(RemoteEvent $event): void
18     {
↕ // ... line 19
20     }
21 }
```

These are the generic mailer remote events Symfony provides. Inside, write `$event->` to see the methods available.

In a real app, this would be where you'd do something with these events like save them to the database or notify an admin if an email bounced. Actually if an email bounces a few times, you may want to update something to *prevent* trying again as this can hurt your email reliability. But for our purposes, just `dump($event)`:

```
src/Webhook/EmailEventConsumer.php
```

```
↕ // ... lines 1 - 11
12 class EmailEventConsumer implements ConsumerInterface
13 {
↕ // ... lines 14 - 16
17     public function consume(RemoteEvent $event): void
18     {
19         dump($event);
20     }
21 }
```

Asynchronous Consumers

One last thing: the webhook controller sends the remote event to the consumer via Symfony Messenger, inside of a message class called `ConsumeRemoteEventMessage`.

To handle this asynchronously & keep your webhook responses fast, in `config/packages/messenger.yaml`, under `routing`, add `Symfony\Component\RemoteEvent\Messenger\ConsumeRemoteEventMessage` and send it to our `async` transport:

```
config/packages/messenger.yaml
```

```
1 framework:
2     messenger:
↕ // ... lines 3 - 11
12     routing:
↕ // ... lines 13 - 15
16         'Symfony\Component\RemoteEvent\Messenger\ConsumeRemoteEventMessage' :
            async
↕ // ... lines 17 - 25
```

Ok! We're ready to demo this webhook. That's next!

Chapter 20: Demoing our Webhook via a Wormhole

Time to test-drive the Mailtrap webhook!

First, we need to switch our development environment to send in production again. In `.env.local`, switch to your production Mailtrap `MAILER_DSN` and in `config/services.yaml`, make sure the `global_from_email`'s domain is the one you configured with Mailtrap.

Create a Webhook on Mailtrap

Over in Mailtrap, go to "Settings" > "Webhooks" and click "Create New Webhook". First thing we need is a Webhook URL. Hmm, this needs to be `/webhook/mailtrap` but it needs to be an absolute URL. In production, this wouldn't be a problem: it would be your production domain. In development, it's a bit trickier. We can't just use the URL the Symfony CLI server gives us...

ngrok

Somehow we need to *expose* our local Symfony server to the public. And there's a neat tool that does exactly this: ngrok. Create a free account, log in, and follow the instructions to configure the ngrok CLI client.

Over in the terminal, restart with Symfony webserver:



```
symfony server:stop
```

Oh, it isn't running. Start it with:



```
symfony serve -d
```

Expose the Local Server

This is the URL we need to expose, copy it and run:

```
ngrok http <paste-url>
```

Paste the URL, and hit enter. Wormhole open!

This crazy looking "Forwarding" URL is the public URL. Copy and paste it into your browser. This warning just lets you know you're running through a tunnel. Click "Visit Site" to see your app. Cool!

Mailtrap Webhook URL

Back in Mailtrap, paste this URL and add `/webhook/mailtrap` to the end. For "Select Stream", choose "Transactional". For "Select Domain", choose your configured Mailtrap domain. Go nuts and select *all* events, then "Save".

Go back into the new webhook and click "Run Test".

"Webhook URL test completed successfully"

That's a good sign!

Dump Server

Remember in our `EmailEventConsumer`, we're just dumping the event? Since hitting the webhook happens behind the scenes, we can't see the dump... or can we? In a new terminal run:

```
symfony console server:dump
```

This hooks into our app and any dumps will be output here live. Clever!

In your browser, book a trip, remember to use a real email address (but not mine!)

MailerDeliveryEvent

Moment of truth! Back in the terminal running the dump server, wait a bit... Alright! We have a dump! Scroll up a bit... This is a `MailerDeliveryEvent` for `delivered`. We see the internal ID Mailtrap assigned, the raw payload, date, recipient email, even our custom metadata and tag.

MailerEngagementEvent

Let's try an engagement event! In your email client, open the email.

Back in the dump server terminal, wait a bit... and boom! Another event! This time, it's a `MailerEngagementEvent` for `open`. This is cool!

Alright space cadets, that's it for this course! We managed to cover almost all of Symfony Mailer features without SPAM'ing our users. Win!

'Til next time, happy coding!

Chapter 21: Bonus: Scheduling our Email Command

Hey! You're still here? Great! I have a bonus chapter for you.

One of our interns, Hugo, is complaining that he has to log in to our server and run the booking reminders command, every night at midnight. I don't know what the problem is - isn't that what interns are for?!

Installing Symfony Scheduler

But... I guess to be more robust, we should automate this in case he's sick or forgets. We could set up a CRON job... but that wouldn't be nearly as cool or flexible as using the Symfony Scheduler component. It's perfect for this. At your terminal, run:

```
composer require scheduler
```

Think of Symfony Scheduler as an add-on for Messenger. It provides its own special transport that, instead of a queue, determines if it's time to run a job. Each job, or task, is a messenger message, so it requires a message handler. You consume the schedule, like any messenger transport with the `messenger:consume` command.

make:schedule

Create a schedule with:

```
symfony console make:schedule
```

Note

`symfony/scheduler` now has an official recipe that creates `src/Schedule.php` for you, so this step is no longer required.

Transport name? Use `default`. Schedule name? Use the default: `MainSchedule`. Exciting!

It's possible to have multiple schedules, but for most apps, a single schedule is enough.

Configuring the Schedule

Check it out: `src/Scheduler/MainSchedule.php`. It's a service that implements `ScheduleProviderInterface` and is marked with the `#[AsSchedule]` attribute with the name `default`. The maker automatically injected the cache, and we'll see why in a second. The `getSchedule()` method is where we configure the schedule and add tasks.

This `->stateful()` that we're passing `$this->cache` to is important. If the process that's running this schedule goes down - like our messenger workers stop temporarily during a server restart - when it comes back online, it will know all the jobs it missed and run them. If a task was supposed to run 10 times while it was down, it will run them all. That might not be desired so add `->processOnlyLastMissedRun(true)` to only run the last one:

```
src/Scheduler/MainSchedule.php
↕ // ... lines 1 - 12
13 final class MainSchedule implements ScheduleProviderInterface
14 {
↕ // ... lines 15 - 19
20     public function getSchedule(): Schedule
21     {
22         return (new Schedule())
↕ // ... lines 23 - 29
30         ->processOnlyLastMissedRun(true)
31         ;
32     }
33 }
```

Bulletproof!

For more complex apps, you might be consuming the same schedule on multiple workers. Use `->lock()` to configure a lock so that only one worker runs the task when its due.

Adding a Task

Time to add our first task! In `->add()`, write `RecurringMessage::`. There are a few different ways to *trigger* a task. I like to use `cron()`. I want this task to run at midnight, every day, so use `0 0 * * *`. The second argument is the messenger message to dispatch. We want to run the `SendBookingRemindersCommand`, but we can't add it here directly. Instead, use `new RunCommandMessage()` and pass the command name: `app:send-booking-reminders` (you can pass arguments and options here too):

```
src/Scheduler/MainSchedule.php
↕ // ... lines 1 - 12
13 final class MainSchedule implements ScheduleProviderInterface
14 {
↕ // ... lines 15 - 19
20     public function getSchedule(): Schedule
21     {
22         return (new Schedule())
23             ->add(
24                 RecurringMessage::cron(
25                     '0 0 * * *',
26                     new RunCommandMessage('app:send-booking-reminders')
27                 )
28             )
↕ // ... lines 29 - 30
31         ;
32     }
33 }
```

Debugging the Schedule

At your terminal, list our schedule's tasks by running:

```
symfony console debug:schedule
```

Oh, we have an error.

"You cannot use "CronExpressionTrigger" as the "cron expression" package is not installed"

Easy fix: copy the install command and run it:


```
composer require dragonmantank/cron-expression
```

Cool name! Now run the debug command again:

```
symfony console debug:schedule
```

Here we go, the output's a little wonky on this small screen, but you can see the cron expression, the message (and command), and the next runtime: tonight at midnight.

#[AsCronTask].

There's an alternate to schedule commands. In `MainSchedule::getSchedule()`, delete the `->add()`. Then jump over to our `SendBookingRemindersCommand` and add another attribute: `#[AsCronTask()]` passing: `0 0 * * *`:

```
src/Command/SendBookingRemindersCommand.php
↕ // ... lines 1 - 19
20 #[AsCronTask('0 0 * * *')]
21 class SendBookingRemindersCommand extends Command
↕ // ... lines 22 - 52
```

In your terminal, debug the schedule again to make sure it's still listed:

```
symfony console debug:schedule
```

And it is, pretty neat.

If you have a lot of tasks scheduled at the same time, like midnight, you might see a CPU spike at this time on your server. Unless it's super important that tasks run at a very specific time, you should spread them out. One way to do this of course, is to manually make sure they all have different cron expressions but... that's a bore.

Hashed Cron Expressions

For our `app:send-booking-reminders` command, I don't care when it runs, just that it runs once a day. We can use a *hashed cron expression*. In our expression, replace the 0's with #'s. The # means "pick a random, valid value for this part":

```
src/Command/SendBookingRemindersCommand.php
↕ // ... Lines 1 - 19
20 #[AsCronTask('# # * * *')]
21 class SendBookingRemindersCommand extends Command
↕ // ... Lines 22 - 52
```

Debug the schedule again:

```
symfony console debug:schedule
```

It's set to run at 5:11am. Run the command again:

```
symfony console debug:schedule
```

It's still 5:11am. Ok, so it's not truly random, the values are calculated deterministically based on the message details. In our case, the string `app:send-booking-reminders`. A different command with the same hash expression will have different values.

The Scheduler documentation has all the details on this. There's even aliases for common hashes. For instance, `#midnight` will pick a time between midnight and 3am. Use that for our expression:

```
src/Command/SendBookingRemindersCommand.php
↕ // ... Lines 1 - 19
20 #[AsCronTask('#midnight')]
21 class SendBookingRemindersCommand extends Command
↕ // ... Lines 22 - 52
```

and debug the schedule again:

```
symfony console debug:schedule
```

Oops, a typo, I'll fix that and run again:

```
symfony console debug:schedule
```

It's now scheduled to run every day at 2:11am. Cool!

Running the Schedule

We've configured our schedule, but how do we run it? Remember, schedules are just Messenger transports. The transport name is `scheduler_<schedule_name>`, in our case, `scheduler_default`. Run it with:

```
symfony console messenger:consume scheduler_default
```

On your production server, configure this to run in the background just like a normal messenger worker.

Alright, that's a quick rundown of the Scheduler component. Check out the documentation to learn more about it!

Happy coding and happy scheduling!

Chapter 22: Bonus: Messenger Monitor Bundle

Hey, you're *still* here? Great! Let's do one final bonus chapter!

When you have a bunch of messages and schedules running in the background, it can be hard to know what's happening. Are my workers running? Is my schedule running? And where is it running to? What about failures? I mean, we have logs, but... *logs*. Instead, let's explore a cool bundle that gives us a UI to get some visibility on what's going on with our army of worker robots!

Installation

At your terminal, run:

```
composer require zenstruck/messenger-monitor-bundle
```

It's asking to install a recipe, say yes. Jump back to our IDE and see what was added.

First, a `src/Schedule.php` was added. This is unrelated to this bundle. Since the last chapter, where we added the `Symfony Scheduler`, it now has an official recipe that adds a default schedule. Since we already have one, delete this file.

MessengerMonitorController

A new controller was added: `src/Controller/Admin/MessengerMonitorController.php`. This is a *stub* to enable the bundle's UI. It extends this `BaseMessengerMonitorController` from the bundle and adds a route prefix of `/admin/messenger`. It also adds this `#[IsGranted('ROLE_ADMIN')]` attribute. This is *super* important for your *real* apps. You *only* want site admins to access the UI as it shows sensitive information. We don't have security configured in this app, so I'll just remove this line:

```
src/Controller/Admin/MessengerMonitorController.php
```

```
↕ // ... Lines 1 - 7  
8 #[Route('/admin/messenger')]  
9 class MessengerMonitorController extends BaseMessengerMonitorController  
10 {  
11 }
```

ProcessedMessage

`src/Entity/ProcessedMessage.php` is a new entity added by the recipe. This is also a *stub* that extends this `BaseProcessedMessage` class and adds an ID column. This is used to track the history of your messenger messages. For every message processed, a new one of these entities is persisted. Don't worry though, this is done in your worker process, so it won't slow down your app's frontend.

Since we have a new entity, we *should* be adding a migration, but I don't have migrations configured for this project. So in your terminal, run:

```
symfony console doctrine:schema:update --force
```

Install Optional Dependencies

Before we check out the UI, the bundle has two optional dependencies that I want to install.

First:

```
composer require knplabs/knp-time-bundle
```

This makes the UI's timestamps human-readable - like "4 minutes ago". Next:

```
composer require lorisleiva/cron-translator
```

Since we're using cron expressions for our scheduled tasks, this package makes them human-readable. So instead of "11 2 * * *", it will display this as "every day at 2:11 AM". Slick!

We're ready to go! Start the server with:

A terminal window with a dark blue header bar containing three white circles. The main area is light gray and contains the text 'symfony serve -d' in a monospaced font.

```
symfony serve -d
```

Dashboard

Jump over to the browser and visit: `/admin/messenger`. This is the Messenger Monitor dashboard!

This first widget shows running workers and their status. We can see we have 1 worker running for our `async` transport. This is the one we configured to run with our Symfony CLI server.

Below, we see our available transports, how many messages are queued, and how many workers are running them. Notice it shows our `scheduler_default` transport as not running. This is expected, as we didn't configure it to run locally.

Below that, we have a snapshot of statistics for the last 24 hours.

On the right, we will see the last 15 messages processed. This is of course empty right now.

All these widgets autorefresh every 5 seconds.

Schedule

Let's create some history! In the top bar, click on `Schedule` (note the icon is red to further indicate the schedule isn't running). This is kind of a "more advanced `debug:schedule` command". We see our single scheduled task: `RunCommandMessage` for `app:send-booking-reminders`. It uses a `CronExpressionTrigger` to run "every day at 2:11 AM". 0 runs so far but we can run it manually by clicking "Trigger"... and selecting our `async` transport.

"Details"

Jump back to the dashboard. It ran successfully, took 58ms, and consumed 31MB of memory. Click "Details" to see even more information! "Time in Queue", "Time to Handle", timestamps... lots of good stuff.

These tags are super helpful for filtering messages. You can add your own tags but some are added by the bundle: `manual`, because we "manually" ran a scheduled task, `schedule`, because it was a scheduled task, `schedule:default`, because it's part of our *default* schedule. This `schedule:default:<hash>` is the unique ID for this scheduled task.

On the right here is the "result" of the message "handler" - in this case, `RunCommandMessageHandler`. Different handlers have different results (some have none). For this one, the result is the command's exit code and output.

"Sent 0 booking reminders"

Let's run this task again, but this time, with a booking that needs a reminder sent. Back in your terminal, reload our fixtures:

```
symfony console doctrine:fixtures:load
```

Back to the browser. The dashboard is empty now but that's expected: reloading our fixtures also cleared our message history. Click "Schedule", then "Trigger" on our "async" transport.

Back on the dashboard, we have 2 messages now. `RunCommandMessage` again but click its "Details":

"Sent 1 booking reminders"

Now our second message: `SendEmailMessage`. This was dispatched by the command. Click its "Details" to see email-related information for its results. Note the tag, `booking_reminder`. The bundle automatically detected that we were sending an email with a "Mailer" tag, so it added it here.

Transports

In the top menu, you can click "Transports" to see more details on each one's pending messages (if applicable). The `failed` transport shows failed messages and gives you the option to retry or remove them, right from the UI!

History

"History" is where we can filter messages: Period, limit to a specific date range. Transport, limit to a specific transport. Status, show just successes or failures. Schedule, whether to include or exclude messages triggered by a schedule. Message type, filter by message class.

Statistics

"Statistics" shows a per-message-class stat summary and can be limited to a specific date-range.

Purge Message History

As you can probably imagine, if your app executes a lot of messages, our history table can get *really* big. The bundle provides some commands to purge older messages.

In the bundle docs, scroll down to "messenger:monitor:purge" and copy the command. We need to schedule this... but how? With Symfony Scheduler of course! Open

`src/Scheduler/MainSchedule.php` and add a new task with

`->add(RecurringMessage::cron())`. Use `#midnight` so it runs daily between midnight and 3am. Add `new RunCommandMessage()` and paste the command. Add the

`--exclude-schedules` option:


```
src/Scheduler/MainSchedule.php
```

```
↕ // ... lines 1 - 12
13 final class MainSchedule implements ScheduleProviderInterface
14 {
↕ // ... lines 15 - 19
20     public function getSchedule(): Schedule
21     {
22         return (new Schedule())
↕ // ... lines 23 - 24
25         ->add(RecurringMessage::cron(
26             '#midnight',
27             new RunCommandMessage('messenger:monitor:purge --exclude-
schedules'),
28         )
29     )
↕ // ... lines 30 - 34
35     ;
36 }
37 }
```

This will purge messages older than 30 days *except* messages triggered by a schedule. This is important because your scheduled tasks might run once a month or even once a year. This enables you to keep a history of them regardless of their frequency.

Purge Schedule History

We should still clean these up though. So, back in the docs, copy a second command:

`messenger:monitor:schedule:purge`. And in the schedule, add it with

`->add(RecurringMessage::cron('#midnight', new RunCommandMessage()))` and paste:

```
src/Scheduler/MainSchedule.php
```

```
↕ // ... Lines 1 - 12
13 final class MainSchedule implements ScheduleProviderInterface
14 {
↕ // ... Lines 15 - 19
20     public function getSchedule(): Schedule
21     {
22         return (new Schedule())
↕ // ... Lines 23 - 29
30         ->add(RecurringMessage::cron(
31             '#midnight',
32             new RunCommandMessage('messenger:monitor:schedule:purge'),
33         )
34     )
35     ;
36 }
37 }
```

This will purge the history of scheduled messages skipped by the command above *but* keep the last 10 runs of each.

Let's make sure these tasks were added to our schedule. Back in the browser, click "Schedule" and here we go: our two new tasks.

For the task we ran manually earlier, we can see the last run summary, details, and even its history.

Ok friends! That's a quick run-through of the `zenstruck/messenger-monitor-bundle`. Check out the [docs](#) for more information on all its features.

'Til next time, happy monitoring!

With <3 from SymphonyCasts