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.

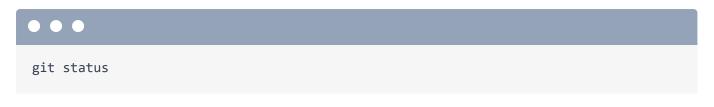
Welcome to "Universal Travel": a travel agency where users can book trips to different galactic locations. Here are the currently available trips. Users *can* already book these, but there are no confirmation emails sent when they do. We're going to fix that! If I'm spending thousands of credits on a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



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:



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
```

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

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

Send the Email

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

```
src/Controller/TripController.php
 1 // ... lines 1 - 18
19 final class TripController extends AbstractController
20 {
 1 // ... lines 21 - 29
        public function show(
30
 1 // ... lines 31 - 35
      ): Response {
36
 1 // ... Lines 37 - 38
           if ($form->isSubmitted() && $form->isValid()) {
39
 1 // ... Lines 40 - 55
56
               $mailer->send($email);
 1 // ... lines 57 - 58
59
           }
 1 // ... 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
 1 // ... lines 1 - 19
20 final class TripController extends AbstractController
21 {
1 // ... Lines 22 - 30
      public function show(
31
1 // ... lines 32 - 36
37 ): Response {
1 // ... lines 38 - 39
          if ($form->isSubmitted() && $form->isValid()) {
40
1 // ... lines 41 - 49
50
               $email = (new Email())
                   ->from(new Address('info@universal-travel.com', 'Universal
51
   Travel'))
1 // ... lines 52 - 54
55
       ;
 1 // ... lines 56 - 59
60
           }
1 // ... 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
1 // ... lines 1 - 19
20 final class TripController extends AbstractController
21 {
1 // ... lines 22 - 30
31 public function show(
1 // ... Lines 32 - 36
37 ): Response {
1 // ... lines 38 - 39
40 if ($form->isSubmitted() && $form->isValid()) {
1 // ... lines 41 - 49
       $email = (new Email())
50
1 // ... line 51
         ->to(new Address($customer->getEmail()))
52
1 // ... lines 53 - 54
    ;
55
1 // ... lines 56 - 59
60 }
1 // ... lines 61 - 65
66 }
67 }
```

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

```
src/Controller/TripController.php
1 // ... lines 1 - 19
20 final class TripController extends AbstractController
21 {
1 // ... lines 22 - 30
31 public function show(
1 // ... Lines 32 - 36
37 ): Response {
1 // ... Lines 38 - 39
40 if ($form->isSubmitted() && $form->isValid()) {
1 // ... lines 41 - 49
       $email = (new Email())
50
1 // ... lines 51 - 52
    ->subject('Booking Confirmation for '.$trip->getName())
53
1 // ... Line 54
    ;
55
1 // ... 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
 1 // ... lines 1 - 19
20 final class TripController extends AbstractController
21 {
1 // ... Lines 22 - 30
       public function show(
31
1 // ... Lines 32 - 36
37 ): Response {
1 // ... Lines 38 - 39
          if ($form->isSubmitted() && $form->isValid()) {
40
1 // ... lines 41 - 49
              $email = (new TemplatedEmail())
50
1 // ... lines 51 - 64
65
           }
1 // ... lines 66 - 70
71 }
72 }
```

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

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

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

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

```
src/Controller/TripController.php
 1 // ... lines 1 - 19
20 final class TripController extends AbstractController
21 {
 1 // ... Lines 22 - 30
       public function show(
31
 1 // ... Lines 32 - 36
    ): Response {
37
 1 // ... Lines 38 - 39
40
          if ($form->isSubmitted() && $form->isValid()) {
 1 // ... lines 41 - 49
               $email = (new TemplatedEmail())
50
 1 // ... lines 51 - 54
55
                   ->context([
                       'customer' => $customer,
56
                        'trip' => $trip,
57
58
                        'booking' => $booking,
59
                   ])
60
               ;
 1 // ... lines 61 - 64
          }
65
 1 // ... 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 <u>https://univeral-travel.com/booking</u> - 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 <u>Mailtrap</u>'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 <u>Mailtrap</u>'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 <u>https://mailtrap.io</u> 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 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 Hey {{ customer.name|split(' ')|first }},
 5
 6 Get ready for your trip to {{ trip.name }}!
 7
   Departure: {{ booking.date | date('Y-m-d') }}
 8
 9
10 
       <a href="{{ url('booking_show', {uid: booking.uid}) }}">
11
           Manage your booking
12
       </a>
13
14 
15
16 
17
       Regards, <br>
       The Universal Travel Team
18
19 
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	<pre>\$email = (new TemplatedEmail())</pre>		
\$	// 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

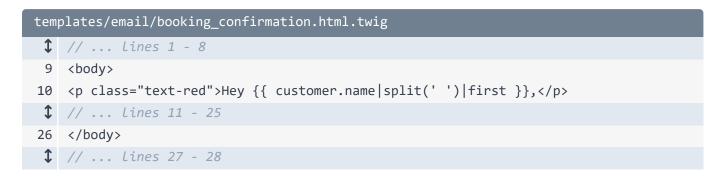
CSS in email requires... some special care. But, pffff, we're Symfony developers! Let's recklessly go forward and see what happens!

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:

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

Now, add this class to the first tag:



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

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

"The style tag is not supported in all email clients."

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

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>
1 {// ... 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:



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 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:



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 <code>@styles/</code> prefix. But wait a minute! <code>email.css</code> 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
   .trip-name {
 1
 2
        font-size: 32px;
   }
 3
 4
 5 .accent-title {
 6
        color: #666666;
 7
   }
 8
 9
   .trip-image {
        border-radius: 12px;
10
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.

\bullet \bullet \bullet

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'),
    source('@styles/email.css')) %}
 2
        <container>
 3
            <row>
 4
               <columns>
 5
                   <spacer size="40"></spacer>
                   Get Ready for your trip to
 6
                   <h1 class="trip-name">{{ trip.name }}</h1>
 7
               </columns>
 8
 9
           </row>
           <row>
10
11
               <columns>
                   Departure: {{ booking.date|date('Y-m-d')
12
    }}
13
               </columns>
14
           </row>
15
           <row>
               <columns>
16
                   <button class="expanded rounded center" href="{{</pre>
17
    url('booking_show', {uid: booking.uid}) }}">
                       Manage Booking
18
19
                   </button>
                   <button class="expanded rounded center secondary" href="{{</pre>
20
    url('bookings', {uid: customer.uid}) }}">
                       My Account
21
                   </button>
22
               </columns>
23
24
           </row>
25
           <row>
26
               <columns>
27
                   We can't wait to see you there,
                   Your friends at Universal Travel
28
               </columns>
29
           </row>
30
31
        </container>
32 {% 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	{		
1	// lines 23 - 31		
32	public function show(
1	// Lines 33 - 38		
39	<pre>#[Autowire('%kernel.project_dir%/assets/terms-of-service.pdf')]</pre>		
40	string \$termsPath,		
41): Response {		
1	// 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
 1 // ... lines 1 - 20
21 final class TripController extends AbstractController
22 {
1 // ... lines 23 - 31
32
       public function show(
1 // ... Lines 33 - 40
      ): Response {
41
 1 // ... Lines 42 - 53
               $email = (new TemplatedEmail())
54
1 // ... lines 55 - 57
58
                  ->attachFromPath($termsPath, 'Terms of Service.pdf')
1 // ... lines 59 - 64
65
        ;
1 // ... lines 66 - 69
70
          }
1 // ... 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:			
1	// 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>
 1 // ... lines 5 - 6
 7
                    <h1 class="trip-name">{{ trip.name }}</h1>
 8
                    <img
                        class="trip-image float-center"
 9
                        src="{{ email.image('@images/%s.png'|format(trip.slug)) }}"
10
                        alt="{{ trip.name }}">
11
12
                </columns>
13
            </row>
 1 // ... lines 14 - 34
35
        </container>
36 {% 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 hal9000@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:

$\bullet \bullet \bullet$

symfony console make:listener

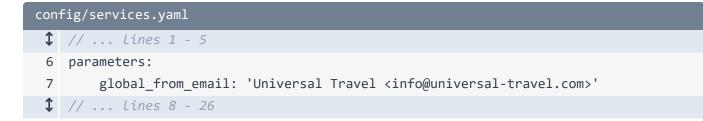
Call it GlobalFromEmailListener. The gives us a list of events we can listen to. We want the first one: MessageEvent. Start typing Symfony and it's autocompleted for us. Hit enter.

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>:



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

final class GlobalFromEmailListener

{
 // ... Lines 12 - 17

 #[AsEventListener]

 public function onMessageEvent(MessageEvent $event): void

  {
  // ... Lines 21 - 31

  }
}
```

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

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
 1 // ... lines 1 - 9
10 final class GlobalFromEmailListener
11 {
 1 // ... lines 12 - 18
       public function onMessageEvent(MessageEvent $event): void
19
       {
20
 1 // ... lines 21 - 22
23
           if (!$message instanceof TemplatedEmail) {
24
               return;
25
           }
 1 // ... 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;:



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

<pre>src/EventListener/GlobalFromEmailListener.php</pre>		
\$	// lines 1 - 9	
10	final class GlobalFromEmailListener	
11	{	
1	// lines 12 - 18	
19	<pre>public function onMessageEvent(MessageEvent \$event): void</pre>	
20	{	
\$	// lines 21 - 30	
31	<pre>\$message->from(\$this->fromEmail);</pre>	
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.

<u>Reply-To</u>

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:



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
1 // ... lines 1 - 21
22 final class TripController extends AbstractController
23 {
1 // ... lines 24 - 32
       public function show(
33
1 // ... lines 34 - 41
    ): Response {
42
1 // ... lines 43 - 44
          if ($form->isSubmitted() && $form->isValid()) {
45
1 // ... lines 46 - 66
               $email->getHeaders()->add(new TagHeader('booking'));
67
1 // ... lines 68 - 71
72
           }
1 // ... 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 keyvalue 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	<pre>if (\$form->isSubmitted() && \$form->isValid()) {</pre>
\$	// lines 47 - 68
69	<pre>\$email->getHeaders()->add(new MetadataHeader('booking_uid', \$booking-</pre>
	>getUid()));
\$	// lines 70 - 74
	}
\$	// lines 76 - 80
81	}
82	}

And:



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:



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.



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:

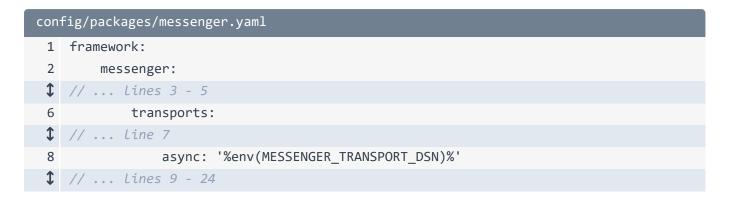
.env	
\$	// lines 1 - 40
41	###> symfony/messenger ###
1	// lines 42 - 44
45	<pre>MESSENGER_TRANSPORT_DSN=doctrine://default?auto_setup=1</pre>
46	###< symfony/messenger ###

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:
1	// 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:

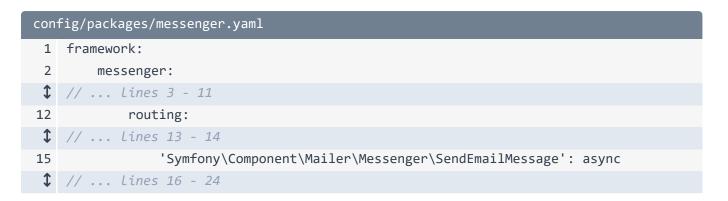


This enables the transport configured with MESSENGER_TRANSPORT_DSN and names it async. It's not obvious here, but failed messages are retried 3 times, with an increasing delay between each attempt. If a message still fails after 3 attempts, it's sent to the failure_transport, called failed, so uncomment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

config/packages/messenger.yaml	
1	framework:
2	messenger:
1	// Lines 3 - 5
6	transports:
1	// Lines 7 - 8
9	<pre>failed: 'doctrine://default?queue_name=failed'</pre>
\$	// 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\SendEmailMessage to the async transport:



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 SendEmailMessage 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:



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: SendEmailMessage. 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!

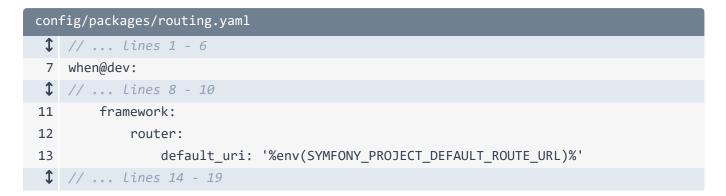
```
config/packages/routing.yaml
1 framework:
2 router:
$ // ... lines 3 - 4
5 default_uri: https://universal-travel.com
$ // ... lines 6 - 19
```

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.

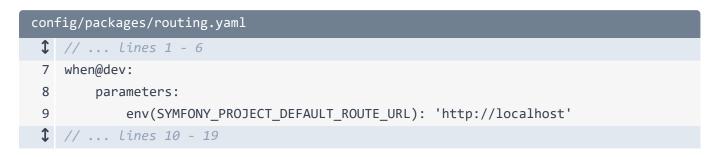
Development Environment Default URL

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)%:



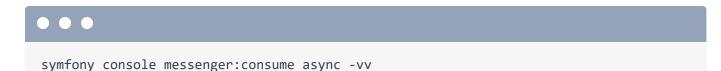
This environment variable will *only* be available if the Symfony CLI server is running *and* you're running commands via symfony console (not bin/console). To avoid an error if the variable is missing, set a default. Still under when@dev, add parameters: with env(SYMFONY PROJECT DEFAULT ROUTE URL):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



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:



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!

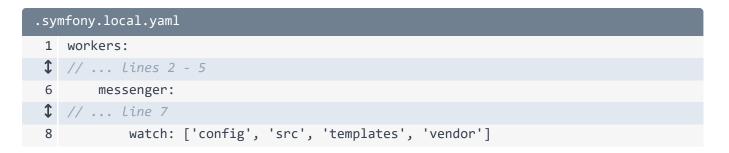
In your IDE, open this .symfony.local.yaml file. This is the Symfony CLI server config for our app. See this workers key? It lets us define processes to run in the background when we start the server. We already have the tailwind command set.

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



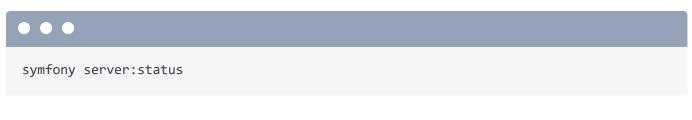
This solves the issue of needing to keep this running in a separate terminal window. But what about restarting the command when we make changes? No problemo! Add a watch key and

set it to config, src, templates and vendor:



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:



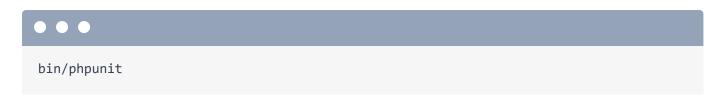
3 workers running! The actual PHP webserver, the existing tailwind:build worker, and our new messenger:consume. So cool!

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:



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	{
1	// lines 15 - 19
20	<pre>public function testCreateBooking(): void</pre>
21	{
1	// Lines 22 - 30
31	<pre>\$this->browser()</pre>
32	->throwExceptions()
\$	// Lines 33 - 42
43	;
\$	// Lines 44 - 52
53	}
54	}

Back in the terminal, run the tests again:

$\bullet \bullet \bullet$

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

Run the tests again!



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.

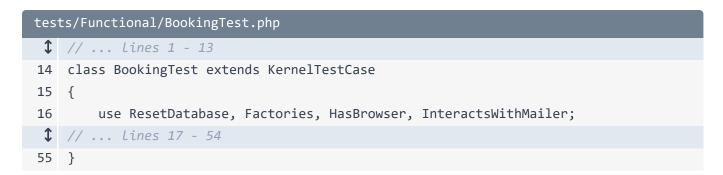
zenstruck/mailer-test

At your terminal, run:

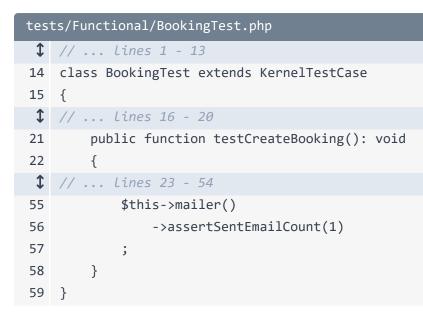
\bullet \bullet \bullet

composer require --dev zenstruck/mailer-test

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



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



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!



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	{
1	// lines 16 - 20
21	<pre>public function testCreateBooking(): void</pre>
22	{
\$	// lines 23 - 54
55	<pre>\$this->mailer()</pre>
\$	// Line 56
57	->assertEmailSentTo('bruce@wayne-enterprises.com', 'Booking
	Confirmation for Visit Mars')
58	;
59	}
60	}

Run the tests!



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

<u>TestEmail</u>

But we can go further! Instead of a string for the subject in this assertion, use a closure with TestEmail \$email as the argument:



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'):
```

test	ts/Functional/BookingTest.php
\$	// lines 1 - 14
15	class BookingTest extends KernelTestCase
16	{
1	// lines 17 - 21
22	<pre>public function testCreateBooking(): void</pre>
23	{
\$	// lines 24 - 55
56	<pre>\$this->mailer()</pre>
\$	// Line 57
58	->assertEmailSentTo('bruce@wayne-enterprises.com', function(TestEmail
	\$email) {
59	\$email
60	->assertSubject('Booking Confirmation for Visit Mars')
1	// lines 61 - 63
64	į
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:

test	ts/Functional/BookingTest.php
\$	// lines 1 - 14
15	class BookingTest extends KernelTestCase
16	{
\$	// lines 17 - 21
22	<pre>public function testCreateBooking(): void</pre>
23	{
1	// lines 24 - 55
56	<pre>\$this->mailer()</pre>
1	// Line 57
58	->assertEmailSentTo('bruce@wayne-enterprises.com', function(TestEmail
	<pre>\$email) {</pre>
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
1 // ... lines 1 - 14
15 class BookingTest extends KernelTestCase
16 {
1 // ... lines 17 - 21
       public function testCreateBooking(): void
22
23
       {
1 // ... lines 24 - 55
56 $this->mailer()
↓ // ... Line 57
              ->assertEmailSentTo('bruce@wayne-enterprises.com', function(TestEmail
58
   $email) {
59
                   $email
1 // ... lines 60 - 61
                       ->assertContains('/booking/'.BookingFactory::first()-
62
   >getUid())
1 // ... 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
 1 // ... lines 1 - 14
15 class BookingTest extends KernelTestCase
16 {
 1 // ... lines 17 - 21
        public function testCreateBooking(): void
22
23
        {
1 // ... lines 24 - 55
          $this->mailer()
56
1 // ... Line 57
               ->assertEmailSentTo('bruce@wayne-enterprises.com', function(TestEmail
58
    $email) {
59
                   $email
 1 // ... lines 60 - 62
                       ->assertHasFile('Terms of Service.pdf')
63
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!

 $\bullet \bullet \bullet$

bin/phpunit

Awesome, 25 assertions now!

<u>->dd()</u>

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
 1 // ... lines 1 - 14
15 class BookingTest extends KernelTestCase
16 {
 1 // ... lines 17 - 21
        public function testCreateBooking(): void
22
23
        {
1 // ... lines 24 - 55
          $this->mailer()
56
1 // ... line 57
               ->assertEmailSentTo('bruce@wayne-enterprises.com', function(TestEmail
58
    $email) {
59
                   $email
1 // ... lines 60 - 63
                       ->dd()
64
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:
1	// Lines 2 - 5
6	messenger:
1	// Line 7
8	<pre>watch: ['config', 'src', 'templates']</pre>

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



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'),
   source('@styles/email.css')) %}
 2
       <container>
           {% block content %}{% endblock %}
 3
 4
           <row>
 5
              <columns>
 6
                   We can't wait to see you there,
 7
                   Your friends at Universal Travel
 8
               </columns>
 9
           </row>
       </container>
10
11 {% 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
   {% extends 'email/layout.html.twig' %}
 1
 2
 3
   {% block content %}
 4
        <row>
 5
            <columns>
 6
                <spacer size="40"></spacer>
 7
                Get Ready for your trip to
                <h1 class="trip-name">{{ trip.name }}</h1>
 8
 9
                <img
                       class="trip-image float-center"
10
                       src="{{ email.image('@images/%s.png'|format(trip.slug)) }}"
11
                       alt="{{ trip.name }}">
12
            </columns>
13
14
        </row>
15
        <row>
16
            <columns>
                Departure: {{ booking.date|date('Y-m-d') }}
17
    </columns>
18
19
        </row>
20
        <row>
            <columns>
21
                <button class="expanded rounded center" href="{{ url('booking_show',</pre>
22
    {uid: booking.uid}) }}">
                   Manage Booking
23
                </button>
24
                <button class="expanded rounded center secondary" href="{{</pre>
25
    url('bookings', {uid: customer.uid}) }}">
                   My Account
26
                </button>
27
            </columns>
28
        </row>
29
30 {% endblock %}
```

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

$\bullet \bullet \bullet$

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. I still looks fantastic!

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!

$\bullet \bullet \bullet$

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
 1 // ... lines 1 - 12
13 class BookingRepository extends ServiceEntityRepository
14 {
 1 // ... lines 15 - 51
      /**
52
        * @return Booking[]
53
        */
54
55
        public function findBookingsToRemind(): array
56
        {
 1 // ... 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():

<pre>src/Repository/BookingRepository.php</pre>	
\$	// lines 1 - 12
13	class BookingRepository extends ServiceEntityRepository
14	{
\$	// lines 15 - 54
55	<pre>public function findBookingsToRemind(): array</pre>
56	{
57	<pre>return \$this->createQueryBuilder('b')</pre>
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,
```

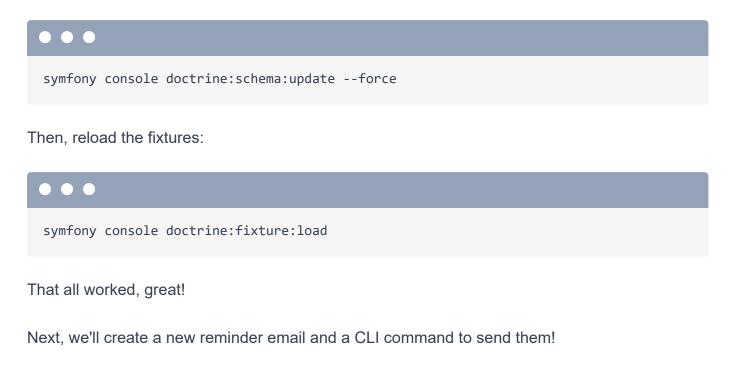
```
'date' => new \DateTimeImmutable('+6 days'):
```

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



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
   {% extends 'email/layout.html.twig' %}
 1
 2
 3
   {% block content %}
 4
        <row>
 5
            <columns>
 6
                <spacer size="40"></spacer>
 7
                Coming soon!
                <h1 class="trip-name">{{ trip.name }}</h1>
 8
 9
                <img
                        class="trip-image float-center"
10
                        src="{{ email.image('@images/%s.png'|format(trip.slug)) }}"
11
                        alt="{{ trip.name }}">
12
            </columns>
13
14
        </row>
15
        <row>
16
            <columns>
                Departure: {{ booking.date|date('Y-m-d') }}
17
    </columns>
18
19
        </row>
20
        <row>
            <columns>
21
                <button class="expanded rounded center" href="{{ url('booking_show',</pre>
22
    {uid: booking.uid}) }}">
                   Manage Booking
23
24
                </button>
25
                <button class="expanded rounded center secondary" href="{{</pre>
    url('bookings', {uid: customer.uid}) }}">
                   My Account
26
27
                </button>
            </columns>
28
29
        </row>
30 {% 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 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":



In the constructor, autowire & set properties for BookingRepository,

```
EntityManagerInterface and MailerInterface:
```

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

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



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:

$\bullet \bullet \bullet$

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

Chapter 18: Email Factory Service

Chapter 19: Webhook for Email Events

Chapter 20: Demo our Webhook

With <3 from SymfonyCasts