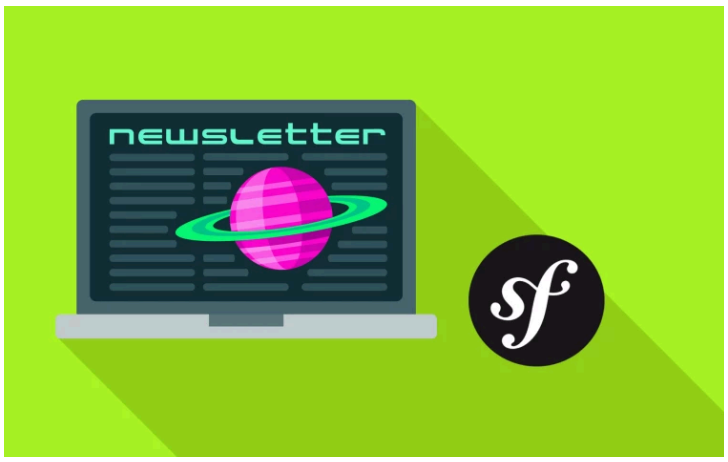
Symfony Mailer: Love Sending Emails Again



Chapter 1: Hello Symfony Mailer

The year is 1995: internet connection speeds are reaching a blistering 56 kbit/s, GeoCities is transforming *everyone* into an accomplished web designer, and sending *emails*... is *all* the rage.

Quick, fast-forward 25 years! Self-driving cars are a reality, you can download an entire HD movie in seconds, we can send rockets into space and then land them safely back on Earth and... yes, love it or hate it... sending emails is *still* all the rage... or at least... something nobody can avoid.

Yep, emails are still a *huge* part of our life and pretty much *every* app needs to send at least some... if not *a lot* of emails. But sending emails has always been kind of a pain - it *feels* like an old process. On top of that, emails are hard to preview, a pain to debug, there are multiple ways to deliver them - do I need an SMTP server? - each email has text *and* HTML parts, and don't even get me *started* about styling emails and embedding CSS in a way that will work in *all* mail clients. Oof.

But then, out of the ashes of this ancient practice grew... a hero. Ok it's actually just a Symfony component - but a cool one! Enter Symfony Mailer: a fresh & modern library that makes something old - sending emails - feel... *new*! Seriously, Mailer actually makes sending emails *fun* again and handles the *ugliest* details automatically. Will you love sending emails after this tutorial? Yea... I think you kinda might!

Setting up the App

As always, unless you're just "mailing it in", you should *totally* code along with me. Dial onto the internet, download the course code from this page and unzip it with WinRAR 1.54b. Inside, you'll find a `start/` directory with the same code that you see here. Open up the `README.md` file to find all the setup details. The *last* step will be to open a terminal, move into the project and use the Symfony Binary to start a web server:

```
symfony serve
```

If you don't have the Symfony binary, you can grab it at Symfony.com/download. Once that's running, open your favorite browser - mine is Netscape Navigator - and go to `https://localhost:8000` to see... The Space Bar! A news site for aliens... and the app that you probably recognize from other Symfony 4 tutorials here on the site.

In this tutorial, we'll be using Symfony 4.3. There *are* a few cool features that are coming in Symfony 4.4 and 5.0... but don't worry! I'll point those out along the way: they aren't big changes, mostly some nice debugging features.

Installing Mailer

Like most things in Symfony, the Mailer component is *not* installed by default. No problem, find your terminal, open a new tab and run:

```
composer require symfony/mailer
```

Notice that I didn't just use `composer require mailer`... using the "mailer" alias. Remember: Symfony Flex lets us say things like `composer require forms` or `composer require templating` and then it maps that to a recommended package. But at the time of this recording, `composer require mailer` would *not* download the Mailer component. Nope, it would download Swift Mailer... was was the recommended library for sending emails with Symfony *before* Symfony 4.3: that's when the Mailer component was introduced.

And even when you're Googling for documentation about Symfony's Mailer, be careful: you might end up on the docs for using *SwiftMailer* inside Symfony. The Mailer docs might be the second or third result.

Anyways after this installs, yea! We get some nice, post-install instructions. We'll talk about *all* of this.

The first step... is to create and configure an Email object! Let's do that next... then send it!

Chapter 2: Creating, Configuring & Sending the Email Object

Time to send... an email! After a user registers for a new account, we should probably send them a welcome email. The controller for this page lives at `src/Controller/SecurityController.php`... find the `register()` method.

This is a very traditional controller: it creates a Symfony form, processes it, saves a new `User` object to the database and ultimately redirects when it finishes.

Let's send an email right here: right *after* the user is saved, but *before* the redirect. How? It's *gorgeous*. Start with `$email = (new Email())` - the one from the `Mime` namespace.

src/Controller/SecurityController.php

```php
// ... lines 1 - 10
use Symfony\Component\Mime\Email;
// ... lines 12 - 16
class SecurityController extends AbstractController
{
// ... lines 19 - 46
    public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guardHandler, LoginFormAuthenticator $formAuthenticator)
    {
// ... lines 49 - 51
        if ($form->isSubmitted() && $form->isValid()) {
// ... lines 53 - 70
            $em->flush();

            $email = (new Email())
// ... lines 74 - 84
        }
// ... lines 86 - 89
    }
}
```

Mime & Mailer Components

Actually, this is a good moment to mention that when we talk about the Mailer component in Symfony, we're actually talking about *two* components: Mailer and Mime. The Mime component is all about creating & configuring the email itself and Mailer is all about *sending* that email. But mostly... that's not too important: just don't be surprised when you're using objects from this `Mime` namespace.

Configuring the Email

I've put the new `Email` object in parentheses on purpose: it allows us to immediately chain off of this to configure the message. Pretty much all the methods on the `Email` class are... delightfully boring & familiar. Let's set the `->from()` address to, how about, `alienmailer@example.com`, the `->to()` to the address of the user that just registered - so `$user->getEmail()` - and this email needs a snazzy subject!

> *"Welcome to the Space Bar!"*

```
src/Controller/SecurityController.php
// ... lines 1 - 10
use Symfony\Component\Mime\Email;
// ... lines 12 - 16
class SecurityController extends AbstractController
{
// ... lines 19 - 46
    public function register(Request $request,
UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler
$guardHandler, LoginFormAuthenticator $formAuthenticator)
    {
// ... lines 49 - 72
            $email = (new Email())
                ->from('alienmailcarrier@example.com')
                ->to($user->getEmail())
                ->subject('Welcome to the Space Bar!')
// ... lines 77 - 89
    }
}
```

Pure poetry. Finally, our email needs content! If you've sent emails before, then you might know that an email can have text content, HTML content *or* both. We'll talk about HTML content soon. But for now, let's set the `->text()` content of the email to:

> *"Nice to meet you"*

And then open curly close curly, `$user->getFirstName()`, and, of course, a ❤️ emoji.

```
src/Controller/SecurityController.php
// ... lines 1 - 10
use Symfony\Component\Mime\Email;
// ... lines 12 - 16
class SecurityController extends AbstractController
{
// ... lines 19 - 46
    public function register(Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guardHandler, LoginFormAuthenticator $formAuthenticator)
    {
// ... lines 49 - 72
            $email = (new Email())
                ->from('alienmailcarrier@example.com')
                ->to($user->getEmail())
                ->subject('Welcome to the Space Bar!')
                ->text("Nice to meet you {$user->getFirstName()}! ❤️");
// ... lines 78 - 89
    }
}
```

There are a bunch more methods on this class, like `cc()`, `addCc()`, `bcc()` and more... but most of these are dead-easy to understand. And because it's such a simple class, you can look inside to see what else is possible, like `replyTo()`. We'll talk about many of these - like attaching files - later.

So... that's it! That's what it looks like to create an email. I hope this "wow'ed" you... and disappointed you in its simplicity... all at the same time.

Sending the Email

Ok... so now... how do we *send* this email? As soon as we installed the Mailer component, Symfony configured a new mailer *service* for us that we can autowire by using - surprise! - the `MailerInterface` type-hint.

Let's add that as one of the arguments to our controller method: `MailerInterface $mailer`.

```
src/Controller/SecurityController.php
// ... lines 1 - 10
use Symfony\Component\Mailer\MailerInterface;
// ... lines 12 - 17
class SecurityController extends AbstractController
{
// ... lines 20 - 47
    public function register(MailerInterface $mailer, Request $request,
UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler
$guardHandler, LoginFormAuthenticator $formAuthenticator)
    {
// ... lines 50 - 92
    }
}
```

And... what methods does this object have on it? Oh, just one: `$mailer->send()` and pass this `$email`.

```
src/Controller/SecurityController.php
// ... lines 1 - 10
use Symfony\Component\Mailer\MailerInterface;
// ... lines 12 - 17
class SecurityController extends AbstractController
{
// ... lines 20 - 47
    public function register(MailerInterface $mailer, Request $request,
UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler
$guardHandler, LoginFormAuthenticator $formAuthenticator)
    {
// ... lines 50 - 52
        if ($form->isSubmitted() && $form->isValid()) {
// ... lines 54 - 73
            $email = (new Email())
                ->from('alienmailcarrier@example.com')
                ->to($user->getEmail())
                ->subject('Welcome to the Space Bar!')
                ->text("Nice to meet you {$user->getFirstName()}! ❤️");

            $mailer->send($email);
// ... lines 81 - 87
        }
// ... lines 89 - 92
    }
}
```

I *love* how this looks. But... will it work? We haven't actually configured *how* emails should be sent but... ah, let's just see what happens. Move over and register: first name `Fox` (last name, Mulder, in case you're wondering), email: `thetruthisoutthere@example.com`, any password, agree to the terms that we definitely read and, register!

Ah! Error!

> *"Environment variable not found: MAILER_DSN"*

Ok, *fine*! To actually *deliver* emails, we need to add some configuration via this environment variable. Let's talk about that next... including some awesome options for debugging emails while you're developing.

Chapter 3: Transport Config & Mailtrap

We've already learned quite a bit about how to customize a specific email... with a *lot* more coming. But how do we customize how an email is *sent*. In Symfony, the way that your messages are delivered is called a *transport*. Go back to your terminal and run:

```
git status
```

The Mailer dsn

When we installed the Mailer component, its *recipe* did a couple of interesting things. First, it created a new file called `config/packages/mailer.yaml`. Let's open up that up. Wow... as you can see: the mailer system doesn't really *have* a lot of config. The only thing here is the `dsn`: a URL that tells Mailer what server or cloud service to use for delivery. This references an environment variable called `MAILER_DSN`. Hey! That's the error we just saw:

> *“Environment variable not found: "MAILER_DSN".”*

The recipe also modified the `.env` file. If you run

```
git diff .env
```

Yep! You'll see that it added a section with an example `MAILER_DSN`.

Configuring MAILER_DSN

Open up `.env`. And, at the bottom, uncomment that `MAILER_DSN` line. By default, this tries to send to a local SMTP server... and I definitely do *not* have one of those running. But... let's try it anyways. Refresh to resubmit the registration form and... boom!

> *“Connection could not be established with host "tcp://localhost:25"”*

So how *are* we going to send emails? Because... there are a *lot* of different options. You could run your own SMTP server... which is not something I recommend... or register with a cloud email sender - like SendGrid - and use your connection details from *them* for Mailer. Mailer supports a *bunch* of the most famous cloud providers... as well as *any* cloud provider that implements SMTP... which is like... all of them. We're going to show how to use SendGrid a bit later.

Why are we not going to use SendGrid right now? Because... when you're developing and debugging your emails, there's a *better* option. Instead of sending *real* emails to a real email server, you can send them to a "fake" mailbox.

One of the most famous tools to do this is called MailCatcher. Basically, you download MailCatcher, start it on your machine, and it creates a temporary SMTP server that you can send to. But instead of *delivering* the messages, it holds onto them and you can view them all in a fake inbox in your browser. MailCatcher is written in Ruby and a similar tool - MailHog - is written in Go. Those are both *great* options.

Hello Mailtrap

But... to save me the headache of getting those running, I'm going to use a *third* option called Mailtrap. Head to mailtrap.io. This is basically a "hosted" version of those tools: it gives us a fake SMTP server and fake inbox, but we don't need to install anything. *And* it has an excellent free plan.

After you register, you'll end up in a spot like this: with a "Demo inbox". Click into that Demo inbox. On the right, you'll see a bunch of information about how to connect to this. At the time of recording, they *do* have specific instructions for Symfony 4... but these are for using Mailtrap with *SwiftMailer*, not Symfony Mailer.

No worries, setup is dead simple. The DSN follows a standard structure: `username:password@server:port`. Copy the username from Mailtrap, paste, add a colon, copy and paste the password, then `@` the server - `smtp.mailtrap.io` - one more colon, and the port. We could use any of these. Try `2525`.

Done! If we haven't messed anything up, our email *should* be delivered to our Mailtrap inbox. Let's try it! Refresh the form submit and... ah! Validation error. The last time we tried this, the email failed to send but the user *was* saved to the database. Make the email unique by adding a "2". Then click the terms, enter any password and... register!

Ok, no errors! Go check Mailtrap! There it is! It's got the subject, *text* content, but no HTML content because we haven't set that yet. There are also a couple of other cool debugging features in Mailtrap - we'll talk about some of these soon.

Now that we've got some success, it's time to attack the obvious shortcoming of this email... it's just text! It's not 1995 anymore people, we need to send *HTML* emails. And Mailer gives us a *great* way to do this: native integration with Twig. That's next.

Chapter 4: HTML Emails with Twig

Every email can contain content in *two* formats, or "parts": a "text" part and an HTML part. And an email can contain *just* the text part, just the HTML part or both. Of course, these days, *most* email clients support HTML, so that's the format you *really* need to focus on. But there *are* still some situations where having a text version is useful - so we won't *completely* forget about text. You'll see what I mean.

The email we just sent did *not* contain the HTML "part" - only the text version. How do we also include an HTML version of the content? Back in the controller, you can almost *guess* how: copy the `->text(...)` line, delete the semicolon, paste and change the method to `html()`. It's that simple! To make it fancier, put an `<h1>` around this.

```
src/Controller/SecurityController.php
// ... lines 1 - 17
class SecurityController extends AbstractController
{
// ... lines 20 - 47
    public function register(MailerInterface $mailer, Request $request,
UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler
$guardHandler, LoginFormAuthenticator $formAuthenticator)
    {
// ... lines 50 - 52
        if ($form->isSubmitted() && $form->isValid()) {
// ... lines 54 - 73
            $email = (new Email())
// ... lines 75 - 77
                ->text("Nice to meet you {$user->getFirstName()}! ❤️")
                ->html("<h1>Nice to meet you {$user->getFirstName()}! ❤️
</h1>");
// ... lines 80 - 88
        }
// ... lines 90 - 93
    }
}
```

This email now has two "parts": a text part and an HTML part. The user's email client will choose which to show, usually HTML. Let's see what this looks like in Mailtrap. Click back to get to the registration form again, change the email address, add a password and... register! No errors! Check out Mailtrap.

Yeah! This time we have an HTML version! One of the things I love about Mailtrap is how easily we can see the original HTML source, the text or the rendered HTML.

MIME: The "Multipart" Behind Emails

Or, you can check what the "Raw" message looks like. Ooooo, nerdy. It turns out that what an email looks like under-the-hood is almost *exactly* what an HTTP response looks like that's returned from our app: it has some headers on top, like `To`, `From` and `Subject`, and *content* below. But, the content *is* a bit different. Normally, our app returns an HTTP response whose *content* is probably HTML or JSON. But this email's content contains *two* formats all at once: HTML *and* text.

Check out the `Content-Type` header: it's `multipart/alternative` and then has this weird `boundary` string - `_=_symfony` - then some random numbers and letters. Below, we can see the content: the plain-text version of the email on top and the `text/html` version below that. That weird `boundary` string is placed between these two... and literally acts as a *separator*: it's how the email client knows where the "text" content stops and the next "part" of the message - the HTML part - begins. Isn't that cool? I mean, if this isn't a hot topic for your next dinner party, I don't know what is.

This is what the Symfony's Mime component helps us build. I mean, sheesh, this is ugly. But all *we* had to do was use the `text()` method to add text content and the `html()` method to add HTML content.

Using Twig

So... as simple as this Email was to build, we're not *really* going to put HTML right inside of our controller. We have our standards! Normally, when we need to write some HTML, we put that in a Twig template. When you need HTML for an email, we'll do the *exact* same thing. Mailer's integration with Twig is *awesome*.

First, if you downloaded the course code, you should have a `tutorial/` directory with a `welcome.html.twig` template file inside. Open up the `templates/` directory. To organize our email-related templates, let's create a new sub-directory called `email/`. Then, paste the `welcome.html.twig` template inside.

templates/email/welcome.html.twig

```
<!doctype html>
<html lang="en">
<head>
// ... lines 4 - 54
</head>
<body>
<div class="body">
    <div class="container">
        <div class="header text-center">
            <a href="#homepage">
                <img src="path/to/logo.png" class="logo" alt="SpaceBar Logo">
            </a>
        </div>
        <div class="content">
            <h1 class="text-center">Nice to meet you %name%!</h1>
            <p class="block">
                Welcome to <strong>the Space Bar</strong>, we can't wait to read what you have to write.
                Get started on your first article and connect with the space bar community.
            </p>
// ... lines 70 - 83
        </div>
// ... lines 85 - 93
    </div>
</div>
</body>
</html>
```

Say hello to our fancy new `templates/email/welcome.html.twig` file. This is a *full* HTML page with embedded styling via a `<style>` tag... and... nothing else interesting: it's 100% static. This `%name%` thing I added here isn't a variable: it's just a reminder of something that we need to make dynamic later.

But first, let's use this! As *soon* as your email needs to leverage a Twig template, you need to change from the `Email` class to `TemplatedEmail`.

Hold Command or Ctrl and click that class to jump into it. Ah, this `TemplatedEmail` class *extends* the normal `Email`: we're really still using the same class as before, but with a few extra methods related to templates. Let's use one of these. Remove *both* the `html()` and `text()` calls - you'll see why in a minute - and replace them with `->htmlTemplate()` and then the normal path to the template: `email/welcome.html.twig`.

```
src/Controller/SecurityController.php
// ... lines 1 - 8
use Symfony\Bridge\Twig\Mime\TemplatedEmail;
// ... lines 10 - 18
class SecurityController extends AbstractController
{
// ... lines 21 - 48
    public function register(MailerInterface $mailer, Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guardHandler, LoginFormAuthenticator $formAuthenticator)
    {
// ... lines 51 - 53
        if ($form->isSubmitted() && $form->isValid()) {
// ... lines 55 - 74
            $email = (new TemplatedEmail())
// ... lines 76 - 77
                ->subject('Welcome to the Space Bar!')
                ->htmlTemplate('email/welcome.html.twig');
// ... lines 80 - 88
        }
// ... lines 90 - 93
    }
}
```

And... that's it! Before we try this, let's make a few things in the template dynamic, like the URLs and the image path. But, there's an important thing to remember with emails: paths must *always* be absolute. That's next.

Chapter 5: Absolute URLs to Routes & Assets

The HTML content of our email will use *this* template... which is still *totally* static. For example, see this link going to `#homepage`? That's just a placeholder. Normally in a template, we would use the `{{ path() }}` function to generate a URL to the homepage route. The name of that route is... check out `ArticleController`... there it is: the homepage route name is `app_homepage`. So we would normally say `path('app_homepage')`.

Using the url() Function

The *problem* is that this will generate a *relative* URL - it will literally generate `href="/"`. But for an email, all paths must be *absolute*. To force that, change `path()` to `url()`.

```
templates/email/welcome.html.twig
<!doctype html>
<html lang="en">
// ... lines 3 - 55
<body>
<div class="body">
    <div class="container">
        <div class="header text-center">
            <a href="{{ url('app_homepage') }}">
// ... line 61
            </a>
        </div>
// ... lines 64 - 93
    </div>
</div>
</body>
</html>
```

That's it! Symfony will detect the domain name - `localhost:8000` while we're coding locally - and use that to prefix the URL.

Let's fix a few other URLs: for the link to create a new article, replace the hardcoded string with `url()` and the name of *that* route, which if you looked in the app, is `admin_article_new`.

```
templates/email/welcome.html.twig
<!doctype html>
<html lang="en">
// ... lines 3 - 55
<body>
<div class="body">
    <div class="container">
// ... lines 59 - 63
        <div class="content">
// ... lines 65 - 69
            <p class="block text-center">
                <a href="{{ url('admin_article_new') }}" class="btn">Get writing!</a>
            </p>
// ... lines 73 - 83
        </div>
// ... lines 85 - 93
    </div>
</div>
</body>
</html>
```

At the bottom, there's one more link to the homepage. Say `{{ url('app_homepage') }}`.

```
templates/email/welcome.html.twig
<!doctype html>
<html lang="en">
// ... lines 3 - 55
<body>
<div class="body">
    <div class="container">
// ... lines 59 - 63
        <div class="content">
// ... lines 65 - 75
            <p class="block text-center">
                <a href="{{ url('app_homepage') }}" class="btn">Get reading!</a>
            </p>
// ... lines 79 - 83
        </div>
// ... lines 85 - 93
    </div>
</div>
</body>
</html>
```

A Bit about Webpack Encore & Images

Links, done! But there's one other path we need to fix: the path to this image. But... forget about emails for a minute. This project uses Webpack Encore to compile its assets: I have an `assets/` directory at the root, an `images` directory inside that, and an `email/logo.png` file that I want to reference. You don't need to run Encore, but if you *did*, I've configured it to *copy* that file into a `public/build/images/` directory. There it is: `public/build/images/email/logo.66125a81.png`.

If you downloaded the starting code for the tutorial, you don't need to worry about running Encore... only because we ran it *for* you and included the final, built `public/build` directory. I mean, you *can* run Encore if you want - you just don't need to because the built files are already there.

The point is, whether you're using Encore or not, the end goal is to generate an absolute URL to a file that lives somewhere in your `public/` directory. To do that in Twig, we use the `{{ asset() }}` function. Pass this `build/images/email/logo.png`. Because we're using Encore, we don't need to include the version hash that's part of the *real* file: the asset function will add that automatically. Go team!

If you're not using Encore, it's the same process: just use `asset()` then include the actual path to the physical file, *relative* to the `public/` directory.

Absolute Image Paths

But... this leaves us with the *same* problem we had for the generated URLs! By default, the `asset()` function generates *relative* URLs: they don't contain the domain name. To fix that, wrap this in another function: `absolute_url()`.

```
templates/email/welcome.html.twig
<!doctype html>
<html lang="en">
// ... lines 3 - 55
<body>
<div class="body">
    <div class="container">
        <div class="header text-center">
            <a href="{{ url('app_homepage') }}">
                <img src="{{ absolute_url(asset('build/images/email/logo.png')) }}" class="logo" alt="SpaceBar Logo">
            </a>
        </div>
// ... lines 64 - 93
    </div>
</div>
</body>
</html>
```

And... done! Ready to try this? Move over to the site, go back, change the email address again... we're going to do this a lot... type a new password, wave a magic wand and... hit enter. Ok... no errors... a good sign!

Over in Mailtrap, it's already there! Oh, it looks *so* much better: we even have a working image and, if we hover over a link, the URL *does* contain our domain: `localhost:8000`. This is even more obvious in the HTML source: everything has a full URL.

Automatic "Text" Part

Woh, and... our email *also* has a text part! How did that happen? In the controller, we *only* called `htmlTemplate()` - we *removed* our call to the `text()` method. Well... thank you Mailer. If you set the HTML on an email but do *not* explicitly set the text, Symfony automatically adds it for you by calling `strip_tags()` on your HTML. That's *awesome*.

Well... awesome... but not *totally* perfect: it included all the styles on top! Don't worry: we'll fix that soon... kinda on accident. But the bottom looks pretty great... with *zero* effort.

Next, the URLs and image paths in our email *are* now dynamic... but nothing else is! Any self-respecting email must have *real* data, like the name of the user... or their favorite color. Let's make the email *truly* dynamic by passing in variables. We'll also find out what *other* information is available for free from inside an email template.

Chapter 6: Email Context & the Magic "email" Variable

When you set the HTML part of an email, Mailer helps out by creating the "text" version for us! It's not perfect... and we'll fix that soon... but... it's a nice start! If you *did* want to control this manually, in `SecurityController`, you could set this the text by calling either the `text()` method or `textTemplate()` to render a template that would only contain text.

Passing Variables (context)

In both cases - `htmlTemplate()` and `textTemplate()` - you're probably going to want to pass some *data* into the template to make the mail dynamic. The way to do this is *not* via a second argument to `htmlTemplate()`. Nope, to pass variables into the templates, call `context()` and give this an `array`. Let's pass a `user` variable set to the `$user` that was just registered.

```
src/Controller/SecurityController.php
// ... lines 1 - 18
class SecurityController extends AbstractController
{
// ... lines 21 - 48
    public function register(MailerInterface $mailer, Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guardHandler, LoginFormAuthenticator $formAuthenticator)
    {
// ... lines 51 - 53
        if ($form->isSubmitted() && $form->isValid()) {
// ... lines 55 - 74
            $email = (new TemplatedEmail())
// ... lines 76 - 79
                ->context([
                    'user' => $user,
                ]);
// ... lines 83 - 91
        }
// ... lines 93 - 96
    }
}
```

As *soon* as we do this, in `welcome.html.twig`, we can replace that weird `%name%` placeholder with `{{ user.firstName }}`... because `user` is a instance of our `User` entity... and it has a `getFirstName()` method on it.

```
templates/email/welcome.html.twig
// ... line 1
<html lang="en">
// ... lines 3 - 55
<body>
<div class="body">
    <div class="container">
// ... lines 59 - 63
        <div class="content">
            <h1 class="text-center">Nice to meet you {{ user.firstName }}!</h1>
// ... lines 66 - 83
        </div>
// ... lines 85 - 93
    </div>
</div>
</body>
</html>
```

Let's try it! In your browser, go back one page, tweak the email, type a password, hit enter and then... there it is! Nice to meet you "Fox".

The Built-in "app" and "email" Variables

But wait, there's more! In addition to whatever variables you pass via `context()`, you *also* have access to exactly two *other* variables... absolutely free. What a deal!

The first one... we already know: it's the `app` variable... which *every* Twig template in Symfony can access. It's useful if you need read info from the session, the request, get the current user or a few other things.

The *other* variable that you magically get access to in all email templates is more interesting. It's called... `emu`. I mean, `email`... and is *not* a large flightless bird from Australia... which would be awesome... but less useful. Nope, it's an an instance of `WrappedTemplatedEmail`.

Hello WrappedTemplatedEmail

I'll hit Shift+Shift and look for `WrappedTemplatedEmail` under "classes".

This is a *super* powerful class... full of *tons* of info. It gives us access to things like the name of *who* the email is being sent to - more about that in a minute - the subject, return path... and it even allows us to *configure* a few things on the email, like embedding an image right from Twig!

We're not going to talk about *all* of these methods... but basically, *any* information about the email itself can be found here... and it even allows you to *change* a few things about the email... all from inside Twig.

Go back to the `welcome.html.twig` email template. All the way at the top, we have a `title` tag set to

> *“Welcome to the Space Bar!”*

Having a `<title>` tag in an email.... is usually not *that* important... but it doesn't hurt to have it and make it match the email's subject. Now that we know about the `email` variable, we can do this properly. Change the text to `{{ email.subject }}`.

```
templates/email/welcome.html.twig
// ... line 1
<html lang="en">
<head>
// ... lines 4 - 5
    <title>{{ email.subject }}</title>
// ... lines 7 - 54
</head>
// ... lines 56 - 96
</html>
```

NamedAddress and email.toName()

Tip

In Symfony 4.4 and higher, you won't see `NamedAddress` mentioned here. But the idea is the same: an address can consist of an email and a "name".

Back inside `WrappedTemplatedEmail`, all the way on top, one of my *favorite* methods is `toName()`. When you're sending an email to just *one* person, this is a *super* nice way to get that person's name. It's interesting... if the "to" is an instance of `NamedAddress`, it returns `$to->getName()`. Otherwise it returns an empty `string`.

What is that `NamedAddress`? Go back to `SecurityController`. Hmm, for the `to()` address... we passed an email *string*... and that's a *totally* legal thing to do. But instead of a string, this method *also* accepts a `NamedAddress` object... or even an *array* of `NamedAddress` objects.

Tip

In Symfony 4.4 and higher, use `new Address()` - it works the same way as the `NamedAddress` we describe here.

Check this out: replace the email string with a `new NamedAddress()`. This takes two arguments: the address that we're sending to - `$user->getEmail()` - *and* the "name" that you want to identify this person as. Let's use `$user->getFirstName()`.

```
src/Controller/SecurityController.php
// ... lines 1 - 13
use Symfony\Component\Mime\NamedAddress;
// ... lines 15 - 19
class SecurityController extends AbstractController
{
// ... lines 22 - 49
    public function register(MailerInterface $mailer, Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guardHandler, LoginFormAuthenticator $formAuthenticator)
    {
// ... lines 52 - 54
        if ($form->isSubmitted() && $form->isValid()) {
// ... lines 56 - 75
            $email = (new TemplatedEmail())
// ... line 77
                ->to(new NamedAddress($user->getEmail(), $user->getFirstName()))
// ... lines 79 - 80
                ->context([
// ... lines 82 - 83
                ]);
// ... lines 85 - 93
        }
// ... lines 95 - 98
    }
}
```

We can do the same thing with from. I'll copy the from email address and replace it with `new NamedAddress()`, `alienmailer@example.com` and for the name, we're sending as `The Space Bar`.

```php
// src/Controller/SecurityController.php
// ... lines 1 - 13
use Symfony\Component\Mime\NamedAddress;
// ... lines 15 - 19
class SecurityController extends AbstractController
{
// ... lines 22 - 49
    public function register(MailerInterface $mailer, Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guardHandler, LoginFormAuthenticator $formAuthenticator)
    {
// ... lines 52 - 54
        if ($form->isSubmitted() && $form->isValid()) {
// ... lines 56 - 75
            $email = (new TemplatedEmail())
                ->from(new NamedAddress('alienmailcarrier@example.com', 'The Space Bar'))
                ->to(new NamedAddress($user->getEmail(), $user->getFirstName()))
// ... lines 79 - 80
                ->context([
// ... lines 82 - 83
                ]);
// ... lines 85 - 93
        }
// ... lines 95 - 98
    }
}
```

This is actually even cooler than it looks... and helps us in *two* ways. First, in `welcome.html.twig`, we can use the `email` object to get the name of the person we're sending to instead of needing the `user` variable.

To prove it, let's get crazy and comment-out the `user` variable in context.

```
src/Controller/SecurityController.php
// ... lines 1 - 13
use Symfony\Component\Mime\NamedAddress;
// ... lines 15 - 19
class SecurityController extends AbstractController
{
// ... lines 22 - 49
    public function register(MailerInterface $mailer, Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guardHandler, LoginFormAuthenticator $formAuthenticator)
    {
// ... lines 52 - 54
        if ($form->isSubmitted() && $form->isValid()) {
// ... lines 56 - 75
            $email = (new TemplatedEmail())
                ->from(new NamedAddress('alienmailcarrier@example.com', 'The Space Bar'))
                ->to(new NamedAddress($user->getEmail(), $user->getFirstName()))
// ... lines 79 - 80
                ->context([
// ... line 82
                    //'user' => $user,
                ]);
// ... lines 85 - 93
        }
// ... lines 95 - 98
    }
}
```

In the template, use `{{ email.toName }}`. This will call the `toName()` method... which *should* give us the first name.

```
templates/email/welcome.html.twig
// ... line 1
<html lang="en">
// ... lines 3 - 55
<body>
<div class="body">
    <div class="container">
// ... lines 59 - 63
        <div class="content">
            <h1 class="text-center">Nice to meet you {{ email.toName }}!
</h1>
// ... lines 66 - 83
        </div>
// ... lines 85 - 93
    </div>
</div>
</body>
</html>
```

This is nice... but the *real* advantage of `NamedAddress` can be seen in the inbox.

Try the flow from the start: find your browser, go back, change the email again - we'll be doing this a lot - type a password, submit and... go check Mailtrap. There it is:

> *“Nice to meet you Fox.”*

It's *now* getting that from the `NamedAddress`. The *real* beauty is on top: from "The Space Bar", then the email and to "Fox" next to that email. This is how pretty much *all* emails you receive appear to come from a specific "name", not just an address.

The "Check HTML"

By the way, one of the tabs in Mailtrap is "Check HTML"... which is kinda cool... well... only "kind of". There is a *lot* of variability on how different email clients *render* emails, like some apparently don't support using the `background-color` style attribute. Crazy!

If you *really* want to test how your emails looks, this "Check HTML" tab probably isn't going to help too much - there are other services like Litmus that can help you. But this *does* highlight one *huge* thing we're doing wrong. It says that some `style` thing on line 7 isn't supported. That's referring to the `style` tag. It turns out that Gmail doesn't support embedding CSS in your email: it doesn't let you do it with a `style` tag *or* with a CSS file. Nope, to make things look good in gmail, you *must* manually put all the styles as style *attributes* on *every* single element. Gross. Fortunately, Mailer will help us with this. We'll see how soon.

But first, let's *perfect* how our auto-generated text content looks... by running one command and high-fiving Mailer.

Chapter 7: Pretty Text Emails

When we send an HTML email, we know that Mailer automatically generates a *text* version for us. Thanks Mailer! And, other than this extra style stuff on top... which we don't really want, it does a pretty good job! But we can make it even *better* - *and* remove those weird extra styles - with one simple command. Find your terminal and run:

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

This is a library that's good at taking HTML and transforming it into Markdown... which, I know, seems like an odd thing to do... but it's super handy! As *soon* as you install it, Mailer will automatically use it to transform the HTML email into text. Well... it will transform the HTML to *markdown*... and it turns out that Markdown is a very attractive text format.

Check it out: on the site, go back, bump the email again, submit and... there's our new email. The HTML looks the same, but check out the text. Yea! *First* of all, the html-to-markdown library was smart enough to get rid of the CSS styles code. It also embedded the logo image on top... which may or may not be useful, but it *does* correctly represent the image & link.

The *most* important thing is that it turned the HTML into a nice structure: the header is obvious, bold content is inside asterisks and the line breaks are correct. Basically, we can now stop worrying about the text emails *entirely*: our emails will have them *and* they will look great.

Next, there are *two* ways to add an image to an email: linking to them or *embedding* them. Let's learn how to embed an image and *when* that's the best option.

Chapter 8: Embedded Images

Look book at the HTML source. When we added the logo earlier, we added it as a normal `img` tag. The only thing special was that we needed to use the `absolute_url` function in Twig to make sure the URL contained our domain.

Linking versus Embedding Images

It turns out that there are *two* ways to put an image into an email. The first is this one: a normal, boring `img` tag that links to your site. The *other* option is to *embed* the image inside the email itself.

There are pros and cons to both. For example, if you link directly to an image on your site... and you delete that image... if the user opens up the email, that image will be broken. But... the fact that you're linking to an image on your site... means that you could *change* the image... and it would change on all the emails.

We'll talk more about *when* you should link to an image versus embed an image in a few minutes. But first, let's see *how* we can *embed* this logo.

Remember, the *source* logo image is located at `assets/images/email/logo.png`. This is the *physical* file we want to embed.

Adding a Twig Path to Images

How do we do that? We're going to do it *entirely* from inside of Twig with a special function that *points* to that image.

But to do this, we need a way to *refer* to the image file from inside of Twig. We're going to do that by adding a new twig *path*. Open up `config/packages/twig.yaml`... and I'll close a few files.

One of the config keys you can put under `twig` is called *paths*... and it's *super* cool. Add one new "path" below this: `assets/images` - I'm literally referring to the `assets/images` directory - set to the word... how about... `images`. That part could be anything.

```yaml
config/packages/twig.yaml
twig:
// ... line 2
    paths:
        'assets/images': images
// ... lines 5 - 9
```

Ok... so *what* did this just do? Forget about emails *entirely* for a minute. Out-of-the-box, when you render a template with Twig, it knows to look for that file in the `templates/` directory... and *only* in the `templates/` directory. If you have template files that live somewhere else, *that* is where "paths" are handy. For example, pretend that, for *some* crazy reason, we decided to put a template inside the `assets/images/` directory called `dark-energy.html.twig`. Thanks to the item we added under `paths`, we could *render* that template by using a special path `@images/dark-energy.html.twig`.

This feature is referred to as "namespaced Twig paths". You configure *any* directory, set it to a string "namespace" - like `images` - then refer to that directory from twig by using `@` then the namespace.

Embedding an Image

In our case, we're not planning to put a *template* inside the `assets/images/` directory and render it. But we *can* leverage the Twig path to refer to the *logo* file.

Back in the template, remove *all* the asset stuff that was pointing to the logo. Replace it with `{{ email.image() }}`. Remember, the `email` variable is an instance of this `WrappedTemplatedEmail` class. We're literally calling this `image()` method: we pass it the physical path to an image file, and it takes care of *embedding* it.

What's the *path* to the logo file? It's `@images/email/logo.png`.

```
templates/email/welcome.html.twig
<!doctype html>
<html lang="en">
// ... lines 3 - 55
<body>
<div class="body">
    <div class="container">
        <div class="header text-center">
            <a href="{{ url('app_homepage') }}">
                <img src="{{ email.image('@images/email/logo.png') }}" class="logo" alt="SpaceBar Logo">
            </a>
        </div>
// ... lines 64 - 93
    </div>
</div>
</body>
</html>
```

Yep, thanks to our config, `@images` points to `assets/images`, and then we put the path after that - `email/logo.png`.

The "cid" and how Images are Embedded

So... what difference does this make in the final email? Let's find out! Go back to the site and do our normal thing to re-submit the registration form. Over in Mailtrap... ok cool - the email *looks* exactly the same. The difference is hiding in the HTML source. Woh! Instead of the image `src` being a URL that points to our site... it's some weird `cid:` then a long string.

This is *great* email nerdery. Check out the "Raw" tab. We already know that the content of the email has multiple parts: here's the text version, below is the `text/html` version and... below *that*, there is now a *third* part of the email content: the logo image! It has a `Content-ID` header - this long `cfdf933` string - and then the image contents below.

The `Content-Id` is the *key*. Inside the message itself, *that* is what the `cid` is referring to. This tells the mail client to go find that "part" of the original message and display it here.

So it's kind of like an email attachment, except that it's displayed *within* the email. We'll talk about *true* email attachments later.

Linking Versus Embedding

So, which method should we use to add images to an email: linking or embedding? Oof, that's a tough question. Embedding an image makes it more robust: if the source image is deleted or your server isn't available, it still shows up. It also makes the email "heavier". This *can* be a problem: if the *total* size of an email gets too big - even 100kb - it *could* start to affect deliverability: a bigger size sometimes counts against your email's SPAM score. Deliverability is an art, but this is something to be aware of.

Some email clients will also make a user click a "Show images from sender" link before displaying *linked* images... but they will display embedded images immediately. But I've also seen some inconsistent handling of embedded images in gmail.

So... the general rule of thumb... if there is one, is this: if you need to include the same image for everyone - like a logo or anything that's part of the email's layout - *link* to the image. But if what you're displaying is *specific* to that email - like the email is showing you a photo that was just shared with your account on the site - theni you can embed the image, if it's small. When you embed, the image doesn't need to be hosted publicly anywhere because it's literally contained *inside* the email.

Next, I already mentioned that the `style` tag doesn't work in gmail... which means that our email will be *completely* unstyled for anyone using gmail. That's... a huge problem. To fix this, *every* style you need *must* be attached directly to the element that needs it via a `style` attribute... which is *insane*! But no worries - Mailer can help, with something called CSS inlining.

Chapter 9: Automatic CSS Inlining

Our email looks good in Mailtrap, but will it look good in Gmail or Outlook? That's one of the things that Mailtrap *can't* answer: it gives us a *ton* of great info about our email... but it is *not* showing an accurate representation of how it would *look* in the real world. If you need to be *super* strict about making sure your email looks good everywhere, check out services like Litmus.

But generally speaking, there are *two* big rules you should follow if you want your emails to display consistently across all mail clients. First, use a *table-based* layout instead of floating or Flex-box. We'll talk about how to do this... without hating it... a bit later. The *second* rule is that you *can't* use CSS files or *even* add a `<style>` tag. These will *not* work in gmail. If you want to style your elements... which you totally *do*... then you literally need to add `style=""` to *every* HTML element.

But... that's insane! It's no way to live! So... we are *not* going to do that. Well... what I mean is, we are not going to do that *manually*.

Checking for the twig-pack

To get this all working, we need to check that a certain bundle is installed. If you started your project *after* October 2019, you can skip this because you *will* already have it.

For older projects, first make sure you have Twig 2.12 or higher: you can find your version by running:

```
composer show twig/twig
```

Mine is too old, so I'll update it by running:

```
composer update twig/twig
```

Now run:

```
composer require twig
```

That... might look confusing: don't we already have Twig installed? Before October 2019, `composer require twig` installed TwigBundle... only. But if you run this command *after* October 16th, 2019 - to be exact - the `twig` alias will download `symfony/twig-pack`. The *only* difference is that the `twig-pack` will install the normal TwigBundle *and* a new `twig/extra-bundle`, which is a library that will help us use some new Twig features. You'll see what I mean.

The *main* point is: make sure `twig/extra-bundle` is installed, and the best way to get it is from the pack. If you installed Twig after October 2019, you probably already have it.

The inline_css Filter

Ok, back to work! In `welcome.html.twig`, *all* the way on top, add `{% apply inline_css %}`.

`inline_css` is actually a *filter*... and in Twig, you *normally* use a filter with the `|` symbol - like `foo|inline_css`. But if you want to run a *lot* of stuff through a filter, you can do it with this handy `apply` *tag*. At the bottom of the template, say `{% endapply %}`.

templates/email/welcome.html.twig

```
{% apply inline_css %}
// ... lines 2 - 98
{% endapply %}
```

And... that's it! This passes our *entire* template through this filter... which is *super* smart. It reads the CSS from inside the `style` tag and uses that to add `style` attributes to every HTML element that it finds. Yea... it's crazy!

Let's see this in action. Go back to `/register` and fill the form back in... I'll use `thetruthisoutthere9@example.com`, any password, agree and... register!

TwigExtraBundle Invites you to Install Packages

It works! I'm kidding! But it's the *next* best thing. The error tells us *exactly* what's going on:

> *“The "inline_css" filter is part of the CssInlinerExtension - try running "composer require twig/cssinliner-extra"”*

Why, what a fabulous idea! This error comes from that new TwigExtraBundle, which allows you to install several outside Twig extension libraries and start using them immediately with zero config. And... to be even *shinier*, if you try to use a feature but don't have the library that the feature requires, it tells you!

Copy the `composer require` line, move over to your terminal, and run:

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

When that finishes... move over to the browser again, hit back and... let's change the email to 9b to be unique. Type a password, hit enter and... go check out that email! It still *looks* the same... but check out the HTML source. The `style` tag *is* still there but if you scroll... *wow*. The styles have been applied to *every* element!

This is one of my absolute *favorite* features of mailer. It's a huge chore that... just works.

Next, let's use this to clean things up even more. Instead of having all this CSS right in the template, let's use a proper, standalone CSS file.

Chapter 10: Inlining CSS Files

Now that the styles are being inlined, we can go a step further. I don't *love* having all my email styles inside a `style` tag. It works... but will be a problem once our app sends *multiple* emails: we don't want to duplicate this in every template.

Nope, in the real world, we put CSS into CSS *files*. Let's do that. Copy *all* of the styles and delete them. Inside the `assets/css` directory, let's create a new `email.css` file. Paste!

assets/css/email.css

```css
body {
    margin: 0;
    padding: 0;
    background-color: #f3f3f3;
    font-family: Helvetica, Arial, sans-serif;
}
h1 {
    background-color: #264459;
    color: #ffffff;
    padding: 30px 0 50px 0;
    font-weight: normal;
}
hr {
    border: none;
    border-top: 3px solid #264459;
    margin: 20px;
}
.container {
    background-color: #fefefe;
    width: 580px;
    margin: 0 auto;
}
.bottom {
    background-color: #efefee;
}
.block {
    margin: 0;
    padding: 10px 20px 20px 20px;
}
.logo {
    width: 100%;
}
.text-center {
    text-align: center;
}
.btn {
    display: inline-block;
    padding: 10px 20px;
    background-color: #264459;
    color: #fefefe;
    border: 1px solid #fff;
    border-radius: 3px;
    font-size: 20px;
    font-weight: bold;
    text-decoration: none;
}
```

So far, we've seen that the `inline_css` filter is smart enough to *notice* any `style` tags in the template and use that CSS to style the HTML tags. But you can *also* point the filter to an *external* CSS file.

Go back to `config/packages/twig.yaml`. To point to the CSS file, we need to add another Twig path: let's set the `assets/css` directory to `styles`. So, `@styles` will point here.

```
config/packages/twig.yaml
twig:
// ... line 2
    paths:
// ... line 4
        'assets/css': styles
// ... lines 6 - 10
```

Back in `welcome.html.twig`, we can pass an argument to `inline_css()`: a *string* of styles that it should use. To get that, use the `source()` function, `@styles/` and then the name of our file `email.css`.

```
templates/email/welcome.html.twig
{% apply inline_css(source('@styles/email.css')) %}
// ... lines 2 - 50
{% endapply %}
```

The `source()` function is a standard Twig function... that you don't see very often. It tells Twig to go find the file - which could be a CSS file or another Twig template - and return its *contents*. It's basically a `file_get_contents()` for Twig. That's perfect, because `inline_css()` doesn't want the *path* to a CSS file, it wants the *string* styles it should use.

Let's try this! Hit back once again in your browser, bump the email, type a password, submit and... it looks good! And *this* time in the HTML source, the `style` tag is *not* there... but the inline *styles* are. That's another benefit of the CSS *file*: it got rid of the extra `style` tag, which makes our email a little bit smaller.

Using Sass or Encore for Email CSS?

By the way, if you prefer to use Sass or LESS for your CSS and are using Webpack Encore to compile all of that into your final CSS file, then... you have a problem. You *must* pass *CSS* to `inline_css` - you can't pass it Sass and expect it to know how to process that. Instead, you need to point `inline_css` at the final, *built* version of your CSS - the file that lives in `public/build/`.

Doing that *seems* easy enough: you could add another Twig path - maybe called `encore` - that refers to the `public/build` directory. Except... if you're using versioned filenames... then how do you know exactly what the built filename will be? And if you're using `splitEntryChunks()`, your *one* CSS file may be split into multiple!

This is a *long* way of saying that pointing to a CSS file with `inline_css` is easy... but pointing to a Sass file is... trickier. Later, we'll walk you through how to do it.

But first! The two rules of making an email look good in every email client are, one, use a table-based layout instead of floats or flex-box. And two, inline your styles. We've done the second, *now* its time to do the first. Does this mean we need to rewrite our HTML to use ugly, annoying tables? Actually... no!

Chapter 11: Ink: Automatic CSS Email Framework

Our email template is HTML... very *traditional* HTML. What I mean is, this is the type of HTML and CSS you would see on a normal website. And, at least inside Mailtrap... it looks good! But a *big* lesson of sending emails is that the HTML is often *not* rendered like a normal browser would render it. Some email clients don't support float or flexbox... so if you're using *those* to establish an email layout then... oof, it's going to look *bad* for some people... like people using gmail.

If you want to write an email that's going to look consistently good in every email client, the best practice is actually to use *tables* for your layout. If you have *no* idea what a table layout is... oh, you are *so*, *so* lucky. Back in the dark ages of the Internet, back before CSS float and flexbox existed, every webpage's layout consisted of tables, rows and cells. It was tables, inside of tables, inside of tables, inside of tables. It was... a nightmare.

So... um... am I saying that the nightmare of needing to write table-based layouts is *still* a reality when you create emails? Yes... and no. Mailer has another trick up its sleeve.

Hello Ink / Foundation for Emails

Google for "Inky Framework" to find something called "Ink" by "Zurb". Let me define... a few things. Zurb is the name of a company, a cool name - it sounds like an alien race: "the Zurb". Anyways, Zurb is the company that created "Foundation": a CSS framework that's probably the second most famous in the world behind Bootstrap. "Ink" is the name of a CSS framework that's designed *specifically* for emails. And actually, they've renamed "Ink" to just "Foundation for Emails".

So, Ink, or Foundation for Emails is a CSS framework for responsive HTML emails that works on any device. Even Outlook! Click on the docs.

Foundation for emails is basically two parts. First, it's a CSS file that defines useful CSS classes and a grid structure for designing emails. Again... it's just like Bootstrap CSS for emails.

The Inky Templating Language

That CSS file is super handy. But the *second* part of Foundation for emails is even *more* interesting. Click the "Inky" link on the left. The *second* part of this library is centered around a custom templating language called "Inky". It's a simple, but *fascinating* tool. Click the "Switch to Inky" link.

Here's the idea: *we* write HTML using some custom Inky HTML tags, like `<container>`, `<row>` and `<columns>`... as well as a few others like `<button>` and `<menu>`. Then, Inky will *transform* this pretty HTML into the crazy, ugly table-based layout required for it to render in an email! Yea, it lets us have table-based emails... without needing to use tables! Yeehaw!

Using the inky_to_html Filter

Now if you downloaded the course code, you should have a `tutorial/` directory, which holds the original `welcome.html.twig` *and* an `inky/` directory with an *updated* `welcome.html.twig`. New stuff!

This is basically the same template but written in that special "Inky" markup: containers, rows, columns, etc. Copy the contents... and let's close a few things. Then open up `templates/email/welcome.html.twig` and *completely* replace this file with the updated version.

It's *really* the same email as before: it has the same dynamic URLs and is printing the recipient's name. It's *just* different markup. Oh, and notice that the `inline_css()` stuff we added a few minutes ago is *gone*! Gasp! Don't worry: we'll put that back in a minute. But until then, forget about CSS.

If we sent this email right now, it would *literally* send with this markup. To *transform* this into the table-based markup we want, we'll use another special filter on the *entire* template. On top, add `{% apply inky_to_html %}`... and *all* the way at the bottom, put `{% endapply %}`. I'll indent this to make it look nice.

templates/email/welcome.html.twig

```
{% apply inky_to_html %}
    <container>
        <row class="header">
            <columns>
                <a href="{{ url('app_homepage') }}">
                    <img src="{{ email.image('@images/email/logo.png') }}" class="logo" alt="SpaceBar Logo">
                </a>
            </columns>
        </row>
        <row class="welcome">
            <columns>
                <spacer size="35"></spacer>
                <h1>
                    <center>
                        Nice to meet you {{ email.toName }}!
                    </center>
                </h1>
                <spacer size="10"></spacer>
            </columns>
        </row>
        <spacer size="30"></spacer>
        <row>
            <columns>
                <p>
                    Welcome to <strong>the Space Bar</strong>, we can't wait to read what you have to write.
                    Get started on your first article and connect with the space bar community.
                </p>
            </columns>
        </row>
        <row>
            <columns>
                <center>
                    <button href="{{ url('admin_article_new') }}">Get writing!</button>
                </center>
            </columns>
        </row>
        <row>
            <columns>
                <p>
                    Check out our existing articles and share your thoughts in the comments!
                </p>
            </columns>
        </row>
        <row>
            <columns>
                <center>
                    <button href="{{ url('app_homepage') }}">Get reading!</button>
                </center>
            </columns>
        </row>
        <row>
            <columns>
                <p>
                    We're so excited that you've decided to join us in our corner of the universe,
                    it's a friendly one with other creative and insightful writers just like you!
                    Need help from a friend? We're always just a message away.
                </p>
            </columns>
        </row>
        <row class="footer">
            <columns>
                <p>Cheers,</p>
                <p>Your friendly <em>Space Bar Team</em></p>
            </columns>
        </row>
        <row class="bottom">
            <columns>
                <center>
                    <spacer size="20"></spacer>
                    <div>
                        Sent with ❤️ from the friendly folks at The Space Bar
                    </div>
                </center>
            </columns>
        </row>
    </container>
{% endapply %}
```

Let's try it! Find your browser and make sure you're on the registration page. Let's register as `thetruthisoutthere11@example.com`, any password, check the terms, register and... error!

Ah, but we know this error! Well, not this *exact* error, but almost! This is Twig telling us that we're trying to use a filter that requires an extra library. Cool! Copy the composer require line, move back over to your terminal, and paste:

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

Tip

Make sure you have XSL extension installed for your PHP to be able to use Inky. To check it - you can run `php -m | grep xsl` in your console and check the output has "xsl".

When that finishes... move back to your browser, go *back* to the registration form, tweak that email and... deep breath... register! I think it worked! Let's go check it out.

There's the new email! Oof, it looks *terrible*... but that's only because it doesn't any CSS yet. Check out the HTML source. So cool: it *transformed* our clean markup into table elements! We just took a *huge* step towards making our emails look good in every email client... without needing to write bad markup.

Inlining the foundation-emails CSS

To get this to *look* good, we need to include some CSS from Foundation for Emails. Go back to the documentation, click on the "CSS Version" link and click download. When you unzip this, you'll find a `foundation-emails.css` file inside. Copy that... and paste it into, how about, the `assets/css` directory.

How do we include this in our email template? We already know how: the `inline_css` filter. But instead of adding *another* apply tag around the entire template, we can piggyback off of inky! Add `|inline_css` and pass this `source()` and the path to the CSS file: `@styles/foundation-emails.css`.

Remember: if you look in `config/packages/twig.yaml`, we set up a path that allows us to say `@styles` to refer to the `assets/css` directory. That's how this path works.

And... I still *do* want to include my custom `email.css` code. Copy the `source()` stuff, add a *second* argument to `inline_css` - you can pass this as *many* arguments of CSS as you want

- and point this at `email.css`.

```
templates/email/welcome.html.twig
{% apply inky_to_html|inline_css(source('@styles/foundation-emails.css'),
source('@styles/email.css')) %}
// ... lines 2 - 76
{% endapply %}
```

That should do it! Oh, but before we try this, back in `tutorial/`, that `inky/` directory *also* holds an `email.css` file. Now that we're using a CSS framework, some of the code in our original `email.css`... just isn't needed anymore! This new `email.css` is basically the same as the original one... but with some extra stuff removed. Copy the code from the file, and paste it over the one in `assets/css`.

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

```css
body {
    margin: 0;
    padding: 0;
    background-color: #f3f3f3;
    font-family: Helvetica, Arial, sans-serif;
}
h1 {
    background-color: #264459;
    color: #ffffff;
    /*padding: 30px 0 50px 0;*/
    font-weight: normal;
}
hr {
    border: none;
    border-top: 3px solid #264459;
    margin: 20px;
}
.welcome {
    background-color: #264459;
}
.bottom {
    background-color: #efefee;
}
.logo {
    width: 100%;
}
.text-center {
    text-align: center;
}
table.button a {
    background-color: #264459;
}
table.button table td {
    background-color: #264459;
    border: 2px solid #264459;
}
```

Ok, time to see the final product! Go back to the registration page, update the email, add a password, enter and... go check out Mailtrap. There it is and... it looks awesome. Well, it looks *exactly* like it did before, but in the HTML source, now that we have a table-based layout, we know this will display more consistently across all email clients. I won't say *perfect*... because you'll need to do some testing - but it's now *much* more likely to look good.

So that's "Foundation for Emails". It's, one, a CSS framework for emails... a lot like Bootstrap for emails... and two, a tool to transform the pretty markup known as Inky into the ugly table-based HTML that the CSS framework styles and that email clients require.

Watch your Email Sizes

Before we keep going, one thing to watch out for *regardless* of how you're styling your emails, is email size. It's *far* from a science, but gmail tends to truncate emails once their size is greater than about 100kb: it hides the rest of the email with a link to see more. Keep that in mind, but more than anything, test your emails to make sure they look good in the real world!

Next, let's bootstrap a console command that will send some emails! It turns out that sending emails in a console command requires an extra trick.

Chapter 12: Let's Make a Console Command!

We've created exactly *one* email... and done some pretty cool stuff with it. Let's introduce a *second* email... but with a twist: instead of sending this email when a user does something on the site - like register - we're going to send this email from a console command. And that... changes a few things.

Let's create the custom console command first. Here's my idea: one of the fields on `User` is called `$subscribeToNewsletter`. In our pretend app, if this field is set to true for an *author* - someone that *writes* content on our site - once a week, via a CRON job, we'll run a command that will email them an update on what they published during the last 7 days.

Making the Command

Let's bootstrap the command... the lazy way. Find your terminal and run:

```
php bin/console make:command
```

Call it `app:author-weekly-report:send`. Perfect! Back in the editor, head to the `src/Command` directory to find... our shiny new console command.

src/Command/AuthorWeeklyReportSendCommand.php

```php
<?php

namespace App\Command;

use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;
use Symfony\Component\Console\Style\SymfonyStyle;

class AuthorWeeklyReportSendCommand extends Command
{
    protected static $defaultName = 'app:author-weekly-report:send';

    protected function configure()
    {
        $this
            ->setDescription('Add a short description for your command')
            ->addArgument('arg1', InputArgument::OPTIONAL, 'Argument description')
            ->addOption('option1', null, InputOption::VALUE_NONE, 'Option description')
        ;
    }

    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $io = new SymfonyStyle($input, $output);
        $arg1 = $input->getArgument('arg1');

        if ($arg1) {
            $io->note(sprintf('You passed an argument: %s', $arg1));
        }

        if ($input->getOption('option1')) {
            // ...
        }

        $io->success('You have a new command! Now make it your own! Pass --help to see your options.');

        return 0;
    }
}
```

Let's start customizing this: we don't need any arguments or options... and I'll change the description:

> *“Send weekly reports to authors.”*

src/Command/AuthorWeeklyReportSendCommand.php

```
// ... lines 1 - 12
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 15 - 25
    protected function configure()
    {
        $this
            ->setDescription('Send weekly reports to authors')
        ;
    }
// ... lines 32 - 46
}
```

The *first* thing we need to do is find *all* users that have this `$subscribeToNewsletter` property set to `true` in the database. To keep our code squeaky clean, let's add a custom repository method for that in `UserRepository`. How about `public function findAllSubscribedToNewsletter()`. This will return an `array`.

src/Repository/UserRepository.php

```
// ... lines 1 - 14
class UserRepository extends ServiceEntityRepository
{
// ... lines 17 - 49
    public function findAllSubscribedToNewsletter(): array
    {
// ... lines 52 - 55
    }
// ... lines 57 - 85
}
```

Inside, return `$this->createQueryBuilder()`, `u` as the alias, `->andWhere('u.subscribeToNewsletter = 1')`, `->getQuery()` and `->getResult()`.

```
src/Repository/UserRepository.php
// ... lines 1 - 14
class UserRepository extends ServiceEntityRepository
{
// ... lines 17 - 49
    public function findAllSubscribedToNewsletter(): array
    {
        return $this->createQueryBuilder('u')
            ->andWhere('u.subscribeToNewsletter = 1')
            ->getQuery()
            ->getResult();
    }
// ... lines 57 - 85
}
```

Above the method, we can advertise that this *specifically* returns an array of `User` objects.

```
src/Repository/UserRepository.php
// ... lines 1 - 14
class UserRepository extends ServiceEntityRepository
{
// ... lines 17 - 46
    /**
     * @return User[]
     */
    public function findAllSubscribedToNewsletter(): array
    {
        return $this->createQueryBuilder('u')
            ->andWhere('u.subscribeToNewsletter = 1')
            ->getQuery()
            ->getResult();
    }
// ... lines 57 - 85
}
```

Autowiring Services into the Command

Back in the command, let's autowire the repository by adding a constructor. This is one of the *rare* cases where we have a parent class... and the parent class has a constructor. I'll go to the Code -> Generate menu - or Command + N on a Mac - and select "Override methods" to override the constructor.

Notice that this added a `$name` argument - that's an argument in the parent constructor - and it *called* the parent constructor. That's important: the parent class needs to set some stuff up. But, we don't need to pass the command name: Symfony already gets that from a static property on our class. Instead, make the first argument: `UserRepository $userRepository`. Hit Alt + Enter and select "Initialize fields" to create that property and set it. Perfect.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 4
use App\Repository\UserRepository;
// ... lines 6 - 12
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 15 - 16
    private $userRepository;

    public function __construct(UserRepository $userRepository)
    {
        parent::__construct(null);

        $this->userRepository = $userRepository;
    }
// ... lines 25 - 46
}
```

Next, in `execute()`, clear *everything* out except for the `$io` variable, which is a nice little object that helps us print things and interact with the user... in a pretty way.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 12
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 15 - 32
    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $io = new SymfonyStyle($input, $output);
// ... lines 36 - 45
    }
}
```

Start with `$authors = $this->userRepository->findAllSubscribedToNewsletter()`.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 12
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 15 - 32
    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $io = new SymfonyStyle($input, $output);

        $authors = $this->userRepository
            ->findAllSubscribedToNewsletter();
// ... lines 39 - 45
    }
}
```

Well, this really returns *all* users... not just authors - but we'll filter them out in a minute. To be extra fancy, let's add a progress bar! Start one with `$io->progressStart()`. Then, foreach over `$authors as $author`, and advance the progress inside.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 12
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 15 - 32
    protected function execute(InputInterface $input, OutputInterface $output)
    {
        $io = new SymfonyStyle($input, $output);

        $authors = $this->userRepository
            ->findAllSubscribedToNewsletter();
        $io->progressStart(count($authors));
        foreach ($authors as $author) {
            $io->progressAdvance();
        }
// ... lines 43 - 45
    }
}
```

Oh, and of course, for `progressStart()`, I need to tell it how *many* data points we're going to advance. Use `count($authors)`. Leave the inside of the `foreach` empty for now, and after, say `$io->progressFinish()`. Finally, for a big happy message, add `$io->success()`

> *“Weekly reports were sent to authors!”*

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 12
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 15 - 32
    protected function execute(InputInterface $input, OutputInterface
$output): int
    {
        $io = new SymfonyStyle($input, $output);

        $authors = $this->userRepository
            ->findAllSubscribedToNewsletter();
        $io->progressStart(count($authors));
        foreach ($authors as $author) {
            $io->progressAdvance();
        }
        $io->progressFinish();

        $io->success('Weekly reports were sent to authors!');

        return 0;
    }
}
```

Brilliant! We're not *doing* anything yet... but let's try it! Copy the command name, find your terminal, and do it!

```
php bin/console app:author-weekly-report:send
```

Super fast!

Counting Published Articles

Inside the `foreach`, the next step is to find all the articles this user published - if any - from the past week. Open up `ArticleRepository`... and add a new method for this - `findAllPublishedLastWeekByAuthor()` - with a single argument: the `User` object. This will return an `array`... of articles: let's advertise that above.

```
src/Repository/ArticleRepository.php
// ... lines 1 - 5
use App\Entity\User;
// ... lines 7 - 16
class ArticleRepository extends ServiceEntityRepository
{
// ... lines 19 - 37
    /**
     * @return Article[]
     */
    public function findAllPublishedLastWeekByAuthor(User $author): array
    {
// ... lines 43 - 49
    }
// ... lines 51 - 73
}
```

The query itself is pretty simple: `return $this->createQueryBuilder()` with `->andWhere('a.author = :author)` to limit to only *this* author - we'll set the `:author` parameter in a second - then `->andWhere('a.publishedAt > :week_ago')`. For the placeholders, call `setParameter()` to set `author` to the `$author` variable, and `->setParameter()` again to set `week_ago` to a `new \DateTime('-1 week')`. Finish with the normal `->getQuery()` and `->getResult()`.

```
src/Repository/ArticleRepository.php
// ... lines 1 - 16
class ArticleRepository extends ServiceEntityRepository
{
// ... lines 19 - 37
    /**
     * @return Article[]
     */
    public function findAllPublishedLastWeekByAuthor(User $author): array
    {
        return $this->createQueryBuilder('a')
            ->andWhere('a.author = :author')
            ->andWhere('a.publishedAt > :week_ago')
            ->setParameter('author', $author)
            ->setParameter('week_ago', new \DateTime('-1 week'))
            ->getQuery()
            ->getResult();
    }
// ... lines 51 - 73
}
```

Boom! Back in the command, autowire the repository via the *second* constructor argument: `ArticleRepository $articleRepository`. Hit Alt + Enter to initialize that field.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 4
use App\Repository\ArticleRepository;
// ... lines 6 - 13
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 16 - 18
    private $articleRepository;

    public function __construct(UserRepository $userRepository, ArticleRepository $articleRepository)
    {
// ... lines 23 - 25
        $this->articleRepository = $articleRepository;
    }
// ... lines 28 - 56
}
```

Down in execute, we can say `$articles = $this->articleRepository->findAllPublishedLastWeekByAuthor()` and pass that `$author`.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 13
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 16 - 35
    protected function execute(InputInterface $input, OutputInterface $output)
    {
// ... lines 38 - 42
        foreach ($authors as $author) {
            $io->progressAdvance();

            $articles = $this->articleRepository
                ->findAllPublishedLastWeekByAuthor($author);
// ... lines 48 - 51
        }
// ... lines 53 - 55
    }
}
```

Phew! Because we're actually querying for *all* users, not everyone will be an author... and even less will have authored some articles in the past 7 days. Let's skip those to avoid sending empty emails: if `count($articles)` is zero, then `continue`.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 13
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 16 - 35
    protected function execute(InputInterface $input, OutputInterface $output)
    {
// ... lines 38 - 42
        foreach ($authors as $author) {
            $io->progressAdvance();

            $articles = $this->articleRepository
                ->findAllPublishedLastWeekByAuthor($author);
            // Skip authors who do not have published articles for the last week
            if (count($articles) === 0) {
                continue;
            }
        }
// ... lines 53 - 55
    }
}
```

By the way, in a real app, where you would have hundreds, thousands or even more users, querying for *all* that have subscribed is *not* going to work. Instead, I would make my query smarter by *only* returning users that are authors or even query for a limited number of authors, keep track of which you've sent to already, then run the command over and over again until everyone has gotten their update. These aren't even the only options. The point is: I'm being a little loose with how much data I'm querying for: be careful in a real app.

Ok, I think we're good! I mean, we're not *actually* emailing yet, but let's make sure it runs. Find your terminal and run the command again:

```
php bin/console app:author-weekly-report:send
```

All smooth. Next... let's actually send an email! And then, fix the duplication we're going to have between our two email templates.

Chapter 13: Using a Base Email Template

We found all the authors that want to receive an update about the articles they wrote during the last 7 days. Now, let's *send* them that update as an email.

If you downloaded the course code, you should have a `tutorial/` directory with an `inky/` directory and a file inside called `author-weekly-report.html.twig`. Copy that and throw it into `templates/email/`.

```
templates/email/author-weekly-report.html.twig
// ... line 1
    <container>
        {# Header #}
        <hr>
        <spacer size="20"></spacer>
        <row>
            <columns>
                <p>
                    What a week {{ email.toName }}! Here's a quick review of what you've been up to on the Space Bar this week
                </p>
            </columns>
        </row>
        <row>
            <columns>
                <table>
                    <tr>
                        <th>#</th>
                        <th>Title</th>
                        <th>Comments</th>
                    </tr>
                    <tr>
                        <td>1</td>
                        <td>Article Title</td>
                        <td>99</td>
                    </tr>
                </table>
            </columns>
        </row>
        <row>
            <columns>
                <center>
                    <spacer size="20"></spacer>
                    <button href="{{ url('app_homepage') }}">Check on the Space Bar</button>
                    <spacer size="20"></spacer>
                </center>
            </columns>
        </row>
        {# Footer #}
    </container>
// ... lines 40 - 41
```

Nice! This template is already written using the Inky markup: the markup that Inky will translate into HTML that will work in any email client. But mostly, other than a link to the homepage and the user's name, this is a boring, empty email: we still need to print the core *content* of the email.

Designing, Configuring & Sending that Email

Let's open up `welcome.html.twig`, steal the `apply` line from here, and paste it on top of the new template. This will translate the markup to Inky *and* inline our CSS. At the bottom, add `endapply`... and I'll indent everything to satisfy my burning inner need for order in the universe!

```
templates/email/author-weekly-report.html.twig
{% apply inky_to_html|inline_css(source('@styles/foundation-emails.css'), source('@styles/email.css')) %}
    <container>
// ... lines 3 - 38
    </container>
{% endapply %}
```

To *send* this email, we know the drill! In the command, start with `$email = (new TemplatedEmail())`, `->from()` and... ah: let's cheat a little.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 6
use Symfony\Bridge\Twig\Mime\TemplatedEmail;
// ... lines 8 - 16
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 19 - 40
    protected function execute(InputInterface $input, OutputInterface $output)
    {
// ... lines 43 - 47
        foreach ($authors as $author) {
// ... lines 49 - 53
            if (count($articles) === 0) {
                continue;
            }

            $email = (new TemplatedEmail())
// ... lines 59 - 67
        }
// ... lines 69 - 71
    }
}
```

Go back to `src/Controller/SecurityController.php`, find the `register()` method and copy *its* `from()` line: we'll probably always send *from* the same user. And yes, we'll learn how *not* to duplicate this later. I'll re-type the "S" on `NamedAddress` and hit tab to add the missing `use` statement on top.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 14
use Symfony\Component\Mime\NamedAddress;
// ... line 16
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 19 - 40
    protected function execute(InputInterface $input, OutputInterface $output)
    {
// ... lines 43 - 47
        foreach ($authors as $author) {
// ... lines 49 - 57
            $email = (new TemplatedEmail())
                ->from(new NamedAddress('alienmailcarrier@example.com', 'The Space Bar'))
// ... lines 60 - 67
        }
// ... lines 69 - 71
    }
}
```

Tip

In Symfony 4.4 and higher, use `new Address()` - it works the same way as the old `NamedAddress`.

Ok, let's finish the rest: `->to()` with `new NamedAddress()` `$author->getEmail()` and `$author->getFirstName()`,

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 14
use Symfony\Component\Mime\NamedAddress;
// ... line 16
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 19 - 40
    protected function execute(InputInterface $input, OutputInterface $output)
    {
// ... lines 43 - 47
        foreach ($authors as $author) {
// ... lines 49 - 57
            $email = (new TemplatedEmail())
                ->from(new NamedAddress('alienmailcarrier@example.com', 'The Space Bar'))
                ->to(new NamedAddress($author->getEmail(), $author->getFirstName()))
// ... lines 61 - 67
        }
// ... lines 69 - 71
    }
}
```

`->subject('Your weekly report on The Space Bar!')` and

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 14
use Symfony\Component\Mime\NamedAddress;
// ... line 16
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 19 - 40
    protected function execute(InputInterface $input, OutputInterface $output)
    {
// ... lines 43 - 47
        foreach ($authors as $author) {
// ... lines 49 - 57
            $email = (new TemplatedEmail())
                ->from(new NamedAddress('alienmailcarrier@example.com', 'The Space Bar'))
                ->to(new NamedAddress($author->getEmail(), $author->getFirstName()))
                ->subject('Your weekly report on the Space Bar!')
// ... lines 62 - 67
        }
// ... lines 69 - 71
    }
}
```

`->htmlTemplate()` to render `email/author-weekly-report.html.twig`.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 14
use Symfony\Component\Mime\NamedAddress;
// ... line 16
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 19 - 40
    protected function execute(InputInterface $input, OutputInterface $output)
    {
// ... lines 43 - 47
        foreach ($authors as $author) {
// ... lines 49 - 57
            $email = (new TemplatedEmail())
                ->from(new NamedAddress('alienmailcarrier@example.com', 'The Space Bar'))
                ->to(new NamedAddress($author->getEmail(), $author->getFirstName()))
                ->subject('Your weekly report on the Space Bar!')
                ->htmlTemplate('email/author-weekly-report.html.twig')
// ... lines 63 - 67
        }
// ... lines 69 - 71
    }
}
```

Do we need to pass any variables to the template? *Technically*... no: the only variable we're using so far is the built-in `email` variable. But we *will* need the articles, so let's call `->context([])`. Pass this an `author` variable... I'm not sure if we'll actually need that... and the `$articles` that this author recently wrote.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 14
use Symfony\Component\Mime\NamedAddress;
// ... line 16
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 19 - 40
    protected function execute(InputInterface $input, OutputInterface
$output)
    {
// ... lines 43 - 47
        foreach ($authors as $author) {
// ... lines 49 - 57
            $email = (new TemplatedEmail())
                ->from(new NamedAddress('alienmailcarrier@example.com',
'The Space Bar'))
                ->to(new NamedAddress($author->getEmail(), $author-
>getFirstName()))
                ->subject('Your weekly report on the Space Bar!')
                ->htmlTemplate('email/author-weekly-report.html.twig')
                ->context([
                    'author' => $author,
                    'articles' => $articles,
                ]);
// ... line 67
        }
// ... lines 69 - 71
    }
}
```

Done! Another beautiful `Email` object. We're a machine! How do we send it? Oh, we know that too: we need the mailer service. Add a *third* argument to the constructor: `MailerInterface $mailer`. I'll do our usual Alt+Enter trick and select "Initialize Fields" to create that property and set it.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 13
use Symfony\Component\Mailer\MailerInterface;
// ... lines 15 - 16
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 19 - 22
    private $mailer;

    public function __construct(UserRepository $userRepository,
ArticleRepository $articleRepository, MailerInterface $mailer)
    {
// ... lines 27 - 30
        $this->mailer = $mailer;
    }
// ... lines 33 - 72
}
```

Back down below, give a co-worker a serious "nod"... as if you're about to take on a task of great gravity... but instead, send an email: `$this->mailer->send($email)`.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 16
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 19 - 40
    protected function execute(InputInterface $input, OutputInterface $output)
    {
// ... lines 43 - 47
        foreach ($authors as $author) {
// ... lines 49 - 57
            $email = (new TemplatedEmail())
                ->from(new NamedAddress('alienmailcarrier@example.com', 'The Space Bar'))
                ->to(new NamedAddress($author->getEmail(), $author->getFirstName()))
                ->subject('Your weekly report on the Space Bar!')
                ->htmlTemplate('email/author-weekly-report.html.twig')
                ->context([
                    'author' => $author,
                    'articles' => $articles,
                ]);
            $this->mailer->send($email);
        }
// ... lines 69 - 71
    }
}
```

Love that. In our fixtures, thanks to some randomness we're using, about 75% of users will be subscribed to the newsletter. Before we run the command, let's make sure the data is fresh... with recent article created dates. Run:

```
php bin/console doctrine:fixtures:load
```

This *should* add enough users and articles that about 1-2 authors will be subscribed to the newsletter *and* have recent articles. Try that command:

```
php bin/console app:author-weekly-report:send
```

Ha! It didn't explode! It found 6 authors... or really, 6 users that are subscribed to the newsletter... but anywhere from 0 to 6 of these might *actually* have recent articles. Spin over to Mailtrap. If you *don't* see any emails - try reloading the fixtures again... just in case you got some bad random data, then re-run the command. Oh, and if you got an error when running the command about too *many* emails being sent, you've hit a limit on Mailtrap. The free plan only allows sending 2 emails each 10 seconds. In that case, ignore the error - because two emails *did* send - or reload your fixtures to hopefully send less emails.

We have exactly *one* email: phew! So... we *rock*! Or do we?

I see a few problems. First, the link to the homepage is broken: it links to `localhost`. *Not* `localhost:8000` - or whatever our *real* domain is - just `localhost`. When you send emails from a console command... your paths break. More on that later.

Base Email Template

The second problem is more obvious... and it's my fault: this email is missing the cool header and footer we had in the other email! Why? Simple: in `welcome.html.twig`, we have a header with a logo on top and a footer at the bottom. In `author-weekly-report.html.twig`? I forgot to put that stuff!

Ok, I *really* did it on purpose: we probably *do* want a consistent layout for every email... but we definitely do *not* want to duplicate that layout in *every* email template.

We know the fix! We do it *all* the time in normal twig: create a base template, a base *email* template. In the `templates/email` directory, add a new file called, how about `emailBase.html.twig`.

And... I'll close a few files. In `welcome.html.twig`, copy that *entire* template and paste in `emailBase`. Then... select the *middle* of the template and delete! We basically want the header, the footer and, in the middle, a block for the content. Add `{% block content %}{% endblock %}`.

templates/email/emailBase.html.twig

```
{% apply inky_to_html|inline_css(source('@styles/foundation-emails.css'), source('@styles/email.css')) %}
    <container>
        <row class="header">
            <columns>
                <a href="{{ url('app_homepage') }}">
                    <img src="{{ email.image('@images/email/logo.png') }}" class="logo" alt="SpaceBar Logo">
                </a>
            </columns>
        </row>

        {% block content %}
        {% endblock %}

        <row class="footer">
            <columns>
                <p>Cheers,</p>
                <p>Your friendly <em>Space Bar Team</em></p>
            </columns>
        </row>
        <row class="bottom">
            <columns>
                <center>
                    <spacer size="20"></spacer>
                    <div>
                        Sent with ❤️ from the friendly folks at The Space Bar
                    </div>
                </center>
            </columns>
        </row>
    </container>
{% endapply %}
```

That block name could be anything. Now that we have *this* nifty template, back in `welcome.html.twig`, life gets simpler. On top, start with `{% extends 'email/emailBase.html.twig' %}`. Then, delete the `apply` and `endapply`, and replace it with `{% block content %}`... and `{% endblock %}`.

templates/email/welcome.html.twig

```
{% extends 'email/emailBase.html.twig' %}

{% block content %}
    <row class="welcome">
        <columns>
            <spacer size="35"></spacer>
            <h1>
                <center>
                    Nice to meet you {{ email.toName }}!
                </center>
            </h1>
            <spacer size="10"></spacer>
        </columns>
    </row>
    <spacer size="30"></spacer>
    <row>
        <columns>
            <p>
                Welcome to <strong>the Space Bar</strong>, we can't wait to read what you have to write.
                Get started on your first article and connect with the space bar community.
            </p>
        </columns>
    </row>
    <row>
        <columns>
            <center>
                <button href="{{ url('admin_article_new') }}">Get writing!</button>
            </center>
        </columns>
    </row>
    <row>
        <columns>
            <p>
                Check out our existing articles and share your thoughts in the comments!
            </p>
        </columns>
    </row>
    <row>
        <columns>
            <center>
                <button href="{{ url('app_homepage') }}">Get reading!</button>
            </center>
        </columns>
    </row>
    <row>
        <columns>
            <p>
                We're so excited that you've decided to join us in our corner of the universe,
                it's a friendly one with other creative and insightful writers just like you!
                Need help from a friend? We're always just a message away.
            </p>
        </columns>
    </row>
{% endblock %}
```

If you're wondering why we don't need the `inky_to_html` and `inline_css` filter stuff anymore, it's because the contents of this template will be put into a block that is *inside* of those same filters. The content *will* go through those filters... but we don't need to worry about adding them in *every* template.

Now we can delete most of the content: all we really need is the welcome row... and down below, we can get rid of the bottom and footer stuff. Celebrate your inner desire for order by *un-indenting* this.

Perfecto! Repeat this beautiful code in `author-weekly-report.html.twig`: `{% extends 'email/emailBase.html.twig' %}`, `{% block content %}` and *all* the way at the bottom, `{% endblock %}`. We can also remove the `container` element... and unindent.

```
templates/email/author-weekly-report.html.twig
{% extends 'email/emailBase.html.twig' %}

{% block content %}
    <hr>
    <spacer size="20"></spacer>
    <row>
        <columns>
            <p>
                What a week {{ email.toName }}! Here's a quick review of what you've been up to on the Space Bar this week
            </p>
        </columns>
    </row>
    <row>
        <columns>
            <table>
                <tr>
                    <th>#</th>
                    <th>Title</th>
                    <th>Comments</th>
                </tr>
                <tr>
                    <td>1</td>
                    <td>Article Title</td>
                    <td>99</td>
                </tr>
            </table>
        </columns>
    </row>
    <row>
        <columns>
            <center>
                <spacer size="20"></spacer>
                <button href="{{ url('app_homepage') }}">Check on the Space Bar</button>
                <spacer size="20"></spacer>
            </center>
        </columns>
    </row>
{% endblock %}
```

That felt *great*! Let's see how it looks: run our weekly report:

```
php bin/console app:author-weekly-report:send
```

And... move back over! Woo! Now *every* email can easily share the same "look".

Next, let's finish the email by making it dynamic. *And*, most importantly, let's figure out why our link paths are broken. You need to be extra careful when you send an email from the command line.

Chapter 14: Router Request Context: Fix Paths in the CLI

We sent the email, but it's missing its core content: info about the articles that each author wrote last week. That's no problem for us: we're already passing an `articles` variable to the template via `context()`. In the template, replace the `<tr>` with `{% for article in articles %}`:

templates/email/author-weekly-report.html.twig

```
// ... lines 1 - 2
{% block content %}
// ... lines 4 - 12
    <row>
        <columns>
            <table>
                <tr>
                    <th>#</th>
                    <th>Title</th>
                    <th>Comments</th>
                </tr>
                {% for article in articles %}
// ... lines 22 - 26
                {% endfor %}
            </table>
        </columns>
    </row>
// ... lines 31 - 39
{% endblock %}
```

add the `<tr>`, a `<td>` and print some data: `{{ loop.index }}` to number the list, 1, 2, 3, 4, etc, `{{ article.title }}` and finally, how about: `{{ article.comments|length }}`.

```
templates/email/author-weekly-report.html.twig
// ... lines 1 - 2
{% block content %}
// ... lines 4 - 12
    <row>
        <columns>
            <table>
                <tr>
                    <th>#</th>
                    <th>Title</th>
                    <th>Comments</th>
                </tr>
                {% for article in articles %}
                <tr>
                    <td>{{ loop.index }}</td>
                    <td>{{ article.title }}</td>
                    <td>{{ article.comments|length }}</td>
                </tr>
                {% endfor %}
            </table>
        </columns>
    </row>
// ... lines 31 - 39
{% endblock %}
```

That's good enough. Double check that by running the command:

```
php bin/console app:author-weekly-report:send
```

And... in Mailtrap... we are good.

Why is the Link Broken

Now let's turn to the glaring, horrible bug in our email! Ah! As I mentioned a few minutes ago, if you hover over the link its, gasp, broken! For some reason, it points to `localhost` not our *real* domain... which is `localhost:8000`. Close, but not right.

Hmm. In the template... yea... that looks right: `{{ url('app_homepage') }}`. Ok, then why - when we click on the link - is it broken?

```
templates/email/author-weekly-report.html.twig
// ... lines 1 - 2
{% block content %}
// ... lines 4 - 30
    <row>
        <columns>
            <center>
// ... line 34
                <button href="{{ url('app_homepage') }}">Check on the Space Bar</button>
// ... line 36
            </center>
        </columns>
    </row>
{% endblock %}
```

We know that the `url()` function tells Symfony to generate an *absolute* URL. And... it *is*. I'll run "Inspect Element" on the broken link button. Check out the `href`: `http://localhost` *not* `localhost:8000`. The *same* thing would happen if you deployed this to production: it would *always* say `localhost`. The URL *is* absolute... it's just wrong!

Why? Think about it: in the registration email - where this *did* work - how did Symfony know what our domain was when it generated the link? Did we configure that somewhere? Nope! When you submit the registration form, Symfony simply looks at what the *current* domain is - `localhost:8000` - and uses *that* for all absolute URLs.

But when you're in a console command, there is no request! Symfony has *no* idea if the code behind this site is deployed to `localhost:8000`, `example.com`, or `lolcats.com`. So, it just guesses `localhost`... which is *totally* wrong... but probably better than guessing `lolcats.com`?

If you're sending emails from the command line - or rendering templates for *any* reason that contain paths - you need to help Symfony: you need to *tell* it what domain to use.

Setting router.request_context

To fix this, start by looking inside our `.env` file. One of our keys here is called `SITE_BASE_URL`.

.env

```
// ... lines 1 - 32

SITE_BASE_URL=https://localhost:8000

// ... lines 36 - 39
```

It is the URL to our app. But, but, but! This is *not* a standard Symfony environment variable and Symfony is *not* currently using this. Nope, this is an environment variable that *we* invented in our file uploads tutorial for a totally different purpose. You can see it used in `config/services.yaml`. It has *nothing* to do with how Symfony generates URLs.

Anyways, to fix the path problem, you need to set two special parameters. The first is `router.request_context.scheme`, which you'll set to `https` or `http`. The other is `router.request_context.host` which, for our local development, will be `localhost:8000`.

Tip

In Symfony 5.1, instead of setting these 2 parameters, you can set 1 new piece of config:

```
# config/packages/routing.yaml
framework:
    router:
        # ...
        default_uri: 'https://example.org/my/path/'
```

Now obviously, we don't want to hardcode these - at least not the second value: it will be different on production. Instead, we need to set these as new environment variables. And... hey! In `.env`, the `SITE_BASE_URL` is *almost* what we need... we just need it to be kind of split into two pieces. Hmm.

Check this out, create two new environment variables: `SITE_BASE_SCHEME` set to `https` and `SITE_BASE_HOST` set to `localhost:8000`.

```
.env
// ... lines 1 - 31
### END CUSTOM VARS

SITE_BASE_SCHEME=https
SITE_BASE_HOST=localhost:8000
// ... lines 36 - 41
```

Back in `services.yaml`, use these values: `%env(SITE_BASE_SCHEME)%` and `%env(SITE_BASE_HOST)%`

```
config/services.yaml
// ... lines 1 - 5
parameters:
// ... lines 7 - 9
    router.request_context.scheme: '%env(SITE_BASE_SCHEME)%'
    router.request_context.host: '%env(SITE_BASE_HOST)%'
// ... lines 12 - 53
```

Cool!

Using Environment Variables... in Environment Variables

The problem is that we now have some duplication. Fortunately, one of the properties of environment variables is that... um... they can contain environment variables! For `SITE_BASE_URL`, set it to `$SITE_BASE_SCHEME` - yep, that's legal - `://` and then `$SITE_BASE_HOST`.

```
.env
// ... lines 1 - 31
### END CUSTOM VARS

SITE_BASE_SCHEME=https
SITE_BASE_HOST=localhost:8000
SITE_BASE_URL=$SITE_BASE_SCHEME://$SITE_BASE_HOST
// ... lines 37 - 41
```

I *love* that trick. Anyways, now that we've set those two parameters, Symfony will use *them* to generate the URL instead of trying to guess it.

Tip

This works, but if you need to override the scheme or host in `.env.local`, you would also need to repeat the `SITE_BASE_URL=` to set it again. A better solution would be to set the `SITE_BASE_URL` just once using a config trick in `services.yaml`:

```
parameters:
    env(SITE_BASE_URL): '%env(SITE_BASE_SCHEME)%://%env(SITE_BASE_HOST)%'
```

Try the command one last time:

```
php bin/console app:author-weekly-report:send
```

And... check it out in Mailtrap! Yes! *This* time the link points to `localhost:8000`.

Next! Let's talk about attaching files to an email. Hmm, but to make it more interesting, let's *first* learn how to generate a styled PDF.

Chapter 15: PDF: Snappy, wkhtmltopdf & Template Setup

How can we make the email we're sending from the console command *cooler*? By adding an attachment! Wait, hmm. That's probably *too* easy - Mailer makes attachments simple. Ok, then... how about this: in addition to having the table inside the email that summarizes what the author wrote during the past week, let's generate a PDF with a similar table and attach *it* to the email.

So that's the first challenge: generating a styled PDF... and hopefully enjoying the process!

Installing Snappy & wkhtmltopdf

My favorite tool for creating PDFs is called Snappy. Fly over to your terminal and install it with:

```
composer require "knplabs/knp-snappy-bundle:^1.6"
```

Snappy is a wrapper around a command-line utility called `wkhtmltopdf`. It has some quirks, but is a *super* powerful tool: you create some HTML that's styled with CSS, give it to `wkhtmltopdf`, it *renders* it like a browser would, and gives you back a PDF version. Snappy makes working with `wkhtmltopdf` pretty easy, but you'll need to make sure it's installed on your system. I installed it on my Mac via `brew`.

```
wkhtmltopdf --version
```

Also, check *where* it's installed with `which` or `whereis`:

```
which wkhtmltopdf
```

Mine is installed at `/usr/local/bin/wkhtmltopdf`. If your binary live somewhere else, you'll need to tweak some config. When we installed the bundle, the bundle's recipe added a new section to the bottom of our `.env` file with two new environment variables.

```
.env
// ... lines 1 - 41
###> knplabs/knp-snappy-bundle ###
WKHTMLTOPDF_PATH=/usr/local/bin/wkhtmltopdf
WKHTMLTOIMAGE_PATH=/usr/local/bin/wkhtmltoimage
###
```

These are both used inside a new `knp_snappy.yaml` file that was *also* added by the bundle.

```
config/packages/knp_snappy.yaml
knp_snappy:
    pdf:
        enabled:    true
        binary:     '%env(WKHTMLTOPDF_PATH)%'
        options:    []
    image:
        enabled:    true
        binary:     '%env(WKHTMLTOIMAGE_PATH)%'
        options:    []
```

The `WKHTMLTOPDF_PATH` variable already equals what I have on my machine. So if *your* path is different, copy this, paste it to your `.env.local` file, and customize it. Oh, and don't worry about `wkhtmltoimage`: we won't use that utility.

Creating the PDF Templates

Ultimately, to create the PDF, we're going to render a template with Twig and pass the HTML from that to Snappy so it can do its work. Open up `templates/email/author-weekly-report.html.twig`.

```
templates/email/author-weekly-report.html.twig
{% extends 'email/emailBase.html.twig' %}

{% block content %}
    <hr>
    <spacer size="20"></spacer>
    <row>
        <columns>
            <p>
                What a week {{ email.toName }}! Here's a quick review of what you've been up to on the Space Bar this week
            </p>
        </columns>
    </row>
    <row>
        <columns>
            <table>
                <tr>
                    <th>#</th>
                    <th>Title</th>
                    <th>Comments</th>
                </tr>
                {% for article in articles %}
                <tr>
                    <td>{{ loop.index }}</td>
                    <td>{{ article.title }}</td>
                    <td>{{ article.comments|length }}</td>
                </tr>
                {% endfor %}
            </table>
        </columns>
    </row>
    <row>
        <columns>
            <center>
                <spacer size="20"></spacer>
                <button href="{{ url('app_homepage') }}">Check on the Space Bar</button>
                <spacer size="20"></spacer>
            </center>
        </columns>
    </row>
{% endblock %}
```

Hmm. In theory, we *could* just render *this* template and use its HTML. But... that won't work because it relies on the special `email` variable. And more importantly, we probably don't want the PDF to look *exactly* like the email - we don't want the logo on top, for example.

No problem: let's do some organizing! Copy the table code. Then, in the `templates/email` directory, I'll create a new file called `_report-table.html.twig` and paste!

templates/email/_report-table.html.twig

```
<table>
    <tr>
        <th>#</th>
        <th>Title</th>
        <th>Comments</th>
    </tr>
    {% for article in articles %}
        <tr>
            <td>{{ loop.index }}</td>
            <td>{{ article.title }}</td>
            <td>{{ article.comments|length }}</td>
        </tr>
    {% endfor %}
</table>
```

Let's make this fancier by adding `class="table table-striped"`. Oo, fancy!

templates/email/_report-table.html.twig

```
<table class="table table-striped">
// ... lines 2 - 13
</table>
```

Those CSS classes come from Bootstrap CSS, which our *site* uses, but our emails do *not*. So when we render this table in the email, these won't do anything. But my *hope* is that when we generate the PDF, we will *include* Bootstrap CSS and our table will look pretty.

Back in `author-weekly-report.html.twig`, take out that table and just say `{{ include('email/_report-table.html.twig') }}`

```
templates/email/author-weekly-report.html.twig
// ... lines 1 - 2
{% block content %}
// ... lines 4 - 12
    <row>
        <columns>
            {{ include('email/_report-table.html.twig') }}
        </columns>
    </row>
// ... lines 18 - 26
{% endblock %}
```

Now we can create a template that we will render to get the HTML for the PDF. Well, we *could* just render this `_report-table.html.twig` template... but because it doesn't have an HTML body or CSS, it would look... simply awful.

Instead, in `templates/email/`, create a new file: `author-weekly-report-pdf.html.twig`. To add some basic HTML, I'll use a PhpStorm shortcut that I *just* learned! Add an exclamation point then hit "tab". Boom! Thanks Victor!

```
templates/email/author-weekly-report-pdf.html.twig
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport"
          content="width=device-width, user-scalable=no, initial-scale=1.0, maximum-scale=1.0, minimum-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>

</body>
</html>
```

Because we're going to add Bootstrap CSS to this template, let's add a little Bootstrap structure: `<div class="container">`, `<div class="row">` and `<div class="col-sm-12">`.

```
templates/email/author-weekly-report-pdf.html.twig
// ... lines 1 - 11
<body>
    <div class="container">
        <div class="row">
            <div class="col-sm-12">
// ... lines 16 - 18
            </div>
        </div>
    </div>
</body>
// ... lines 23 - 24
```

Inside, how about an `<h1>` with "Weekly Report" and today's date, which we can get with `{{ 'now'|date('Y-m-d') }}`.

```
templates/email/author-weekly-report-pdf.html.twig
// ... lines 1 - 11
<body>
    <div class="container">
        <div class="row">
            <div class="col-sm-12">
                <h1>Weekly report {{ 'now'|date('Y-m-d') }}</h1>
// ... lines 17 - 18
            </div>
        </div>
    </div>
</body>
// ... lines 23 - 24
```

Bring in the table with `{{ include('email/_report-table.html.twig') }}`.

```
templates/email/author-weekly-report-pdf.html.twig
// ... lines 1 - 11
<body>
    <div class="container">
        <div class="row">
            <div class="col-sm-12">
                <h1>Weekly report {{ 'now'|date('Y-m-d') }}</h1>

                {{ include('email/_report-table.html.twig') }}
            </div>
        </div>
    </div>
</body>
// ... lines 23 - 24
```

Adding CSS to the Template

If we *just* rendered this and passed the HTML to Snappy, it *would* work, but would contain *no* CSS styling... so it would look like it was designed in the 90's. If you look in `base.html.twig`, this project uses Webpack Encore. The `encore_entry_link_tags()` function basically adds the base CSS, which includes Bootstrap.

Copy this line, close that template, and add this to the PDF template.

```
templates/email/author-weekly-report-pdf.html.twig
// ... lines 1 - 2
<head>
// ... lines 4 - 9
    {{ encore_entry_link_tags('app') }}
</head>
// ... lines 12 - 24
```

Even if you're not using Encore, the point is that an *easy* way to style your PDF is by bringing in the same CSS that your site uses. Oh, and because our site has a gray background... but I want my PDF to *not* share *that* specific styling, I'll hack in a `background-color: #fff`.

By the way, if our app needed to generate *multiple* PDF files, I would *absolutely* create a PDF "base template" - like `pdfBase.html.twig` - so that every PDF could share the same look and feel. Also, I'm *not* bringing in any JavaScript tags, but you *could* if your JavaScript is responsible for helping render how your page looks.

Ok, we're ready! Next, let's use Snappy to create the PDF, attach it to the email and high-five ourselves. Because celebrating victories is important!

Chapter 16: Lets Generate a PDF!

Let's transform this Twig template into a PDF.

Back in `AuthorWeeklyReportSendCommand`, right before we create the `Email`, *this* is where we'll generate the PDF, so we can attach it. To do that, our command needs *two* new services: `Environment $twig` - yes, it looks weird, but the type-hint to get Twig directly is called `Environment` - and `Pdf $pdf`. That *second* service comes from SnappyBundle.

src/Command/AuthorWeeklyReportSendCommand.php

```php
// ... lines 1 - 6
use Knp\Snappy\Pdf;
// ... lines 8 - 16
use Twig\Environment;
// ... line 18
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 21 - 28
    public function __construct(UserRepository $userRepository, ArticleRepository $articleRepository, MailerInterface $mailer, Environment $twig, Pdf $pdf)
    {
// ... lines 31 - 37
    }
// ... lines 39 - 84
}
```

As a reminder, if you don't know what type-hint to use, you can always spin over to your terminal and run:

```
php bin/console debug:autowiring pdf
```

There it is!

Ok, step 1 is to use Twig to render the template and get the HTML: `$html = $this->twig->render()`. Oh... PhpStorm doesn't like that... because I forgot to add the properties! I'll put my cursor on the new arguments, hit Alt+Enter, and select "Initialize Fields" to create those 2 properties and set them.

```
src/Command/AuthorWeeklyReportSendCommand.php
```

```php
// ... lines 1 - 6
use Knp\Snappy\Pdf;
// ... lines 8 - 16
use Twig\Environment;
// ... line 18
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 21 - 25
    private $twig;
    private $pdf;

    public function __construct(UserRepository $userRepository, ArticleRepository $articleRepository, MailerInterface $mailer, Environment $twig, Pdf $pdf)
    {
// ... lines 31 - 35
        $this->twig = $twig;
        $this->pdf = $pdf;
    }
// ... lines 39 - 84
}
```

Now, back to work: `$this->twig->render()` and pass this the template name - `email/author-weekly-report-pdf.html.twig` - and an array of the variables it needs... which I think is just `articles`. Pass `'articles' => $articles`.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 18
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 21 - 46
    protected function execute(InputInterface $input, OutputInterface $output)
    {
// ... lines 49 - 53
        foreach ($authors as $author) {
// ... lines 55 - 59
            if (count($articles) === 0) {
                continue;
            }


            $html = $this->twig->render('email/author-weekly-report-pdf.html.twig', [
                'articles' => $articles,
            ]);
// ... lines 68 - 79
        }
// ... lines 81 - 83
    }
}
```

To turn that HTML into PDF content, we can say `$pdf = $this->pdf->getOutputFromHtml($html)`.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 18
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 21 - 46
    protected function execute(InputInterface $input, OutputInterface $output)
    {
// ... lines 49 - 53
        foreach ($authors as $author) {
// ... lines 55 - 64
            $html = $this->twig->render('email/author-weekly-report-pdf.html.twig', [
                'articles' => $articles,
            ]);
            $pdf = $this->pdf->getOutputFromHtml($html);
// ... lines 69 - 79
        }
// ... lines 81 - 83
    }
}
```

Cool, right! Behind the scenes, this simple method does a lot: it takes the HTML content, saves it to a temporary file, then executes `wkhtmltopdf` and *points* it at that file. As long as `wkhtmltopdf` is set up correctly... and our HTML generates a nice-looking page, it should work!

If *all* has gone well, the `$pdf` variable will now be a string containing the actual PDF content... which we could do anything with, like save to a file *or* attach to an email. Why, what a wonderful idea!

Adding an Attachment

Adding an attachment to an email... probably looks exactly like you would expect: `->attach()`. The first argument is the file *contents* - so `$pdf`. If you need to attach something *big*, you can also use a file *resource* here - like use `fopen` on a file and pass the file handle so you don't need to read the whole thing into memory. The second argument will be the filename for the attachment. Let's uses `weekly-report-%s.pdf` and pass today's date for the wildcard: `date('Y-m-d')`.

```php
// src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 18
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 21 - 46
    protected function execute(InputInterface $input, OutputInterface $output)
    {
// ... lines 49 - 53
        foreach ($authors as $author) {
// ... lines 55 - 67
            $pdf = $this->pdf->getOutputFromHtml($html);

            $email = (new TemplatedEmail())
// ... lines 71 - 74
                ->context([
// ... lines 76 - 77
                ])
                ->attach($pdf, sprintf('weekly-report-%s.pdf', date('Y-m-d')));
            $this->mailer->send($email);
        }
// ... lines 82 - 84
    }
}
```

Love it! We're ready to try this thing. Find your terminal and run:

```
php bin/console app:author-weekly-report:send
```

As a reminder, even though this *looks* like it's sending to six authors, it's a lie! It's *really* looping over 6 *possible* authors, but only sending emails to those that have written an article within the past 7 days. Because the database fixtures for this project have a bunch of randomness, this might send to 5 users, 2 users... or 0 users. If it doesn't send *any* emails, try reloading your fixtures by running:

```
php bin/console doctrine:fixtures:load
```

If you are *so* lucky that it's sending *more* than 2 emails, you'll get an error from Mailtrap, because it limits sending 2 emails per 10 seconds on the free plan. You can ignore the error or reload the fixtures.

In my case, in Mailtrap... yea! This sent 2 emails. If I click on the first one... it looks good... and it has an attachment! Let's open it up!

Oh... ok... I guess it *technically* worked... but it looks *terrible*. This definitely did *not* have Bootstrap CSS applied to it. The question is: why not?

Next, let's put on our debugging hats, get to the bottom of this mystery, and *crush* this bug.

Chapter 17: Styling PDFs with CSS

Our PDF attachment looks *terrible*. I don't know *why*, but the CSS is *definitely* not working.

Debugging this can be tricky because, even though this was *originally* generated from an HTML page, we can't exactly "Inspect Element" on a PDF to see what went wrong.

So... let's... think about what's happening. The `encore_entry_link_tags()` function creates one or more link tags to CSS files, which live in the `public/build/` directory. But the paths it generates are *relative* - like `href="/build/app.css"`.

We *also* know that the `getOutputFromHtml()` method works by taking the HTML, saving it to a temporary file and then *effectively* loading that file in a browser... and creating a PDF from what it looks like. If you load a random HTML file on your computer into a browser... and that HTML file has a CSS link tag to `/build/app.css`, what would happen? Well, it would look for that file on the *filesystem* - like literally a `/build/` directory at the root of your drive.

That is what's happening behind the scenes. So, the CSS never loads... and the PDF looks like it was designed... well... by me. We can do better.

Making Absolute CSS Paths

Once you understand what's going on, the fix is pretty simple. Replace `{{ encore_entry_link_tags() }}` with `{% for path in encore_entry_css_files('app') %}`.

```
templates/email/author-weekly-report-pdf.html.twig
// ... lines 1 - 2
<head>
// ... lines 4 - 9
    {% for path in encore_entry_css_files('app') %}
// ... line 11
    {% endfor %}
</head>
// ... lines 14 - 26
```

Instead of printing all the link tags for all the CSS files we need, this allows us to loop over them. Inside, add `<link rel="stylesheet" href="">` and then make the path absolute with `absolute_url(path)`.

```
templates/email/author-weekly-report-pdf.html.twig
// ... lines 1 - 2
<head>
// ... lines 4 - 9
    {% for path in encore_entry_css_files('app') %}
        <link rel="stylesheet" href="{{ absolute_url(path) }}">
    {% endfor %}
</head>
// ... lines 14 - 26
```

We saw this earlier: we used it to make sure the path to our logo - before we embedded it - contained the hostname. *Now* when `wkhtmltopdf`, more or less, opens the temporary HTML file in a browser, it will download the CSS from our public site and all *should* be happy with the world.

Let's try it! Run the console command:

```
php bin/console app:author-weekly-report:send
```

Move back over and... I'll refresh Mailtrap... great! 2 new emails. Check the attachment on the first one. It looks perfect! I mean, hopefully you're better at styling than I am... and can make this look *even* better, maybe with a hot-pink background and unicorn Emojis? I'm still working on my vision. The point is: the CSS *is* being loaded.

Let's check the other email to be sure. What? This one looks terrible! The first PDF is good... and the second one... which was generated the *exact* same way... has no styling!? What madness is this!?

Encore: Missing CSS after First PDF?

This is a little gotcha that's specific to Encore. For reasons that are... not that interesting right now - you can ask me in the comments - when you call an Encore Twig function the first time, it returns all the CSS files that you need for the `app` entrypoint. But when we go through the loop the second time, render a second template and call `encore_entry_css_files()` for a second time, Encore returns an empty array. Basically, you can only call an Encore function for an entrypoint once per request... or once per console command execution. Every time after, the method will return nothing.

There's a good reason for this... but it's *totally* messing us up! No worries, once you know what's going on, the fix is pretty simple. Find the constructor and add one more argument - I know, it's getting a bit crowded. It's `EntrypointLookupInterface $entrypointLookup`. I'll do my normal Alt + Enter and select "Initialize fields" to create that property and set it.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 16
use Symfony\WebpackEncoreBundle\Asset\EntrypointLookupInterface;
// ... lines 18 - 19
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 22 - 28
    private $entrypointLookup;

    public function __construct(UserRepository $userRepository, ArticleRepository $articleRepository, MailerInterface $mailer, Environment $twig, Pdf $pdf, EntrypointLookupInterface $entrypointLookup)
    {
// ... lines 33 - 39
        $this->entrypointLookup = $entrypointLookup;
    }
// ... lines 42 - 88
}
```

Down below, right before we render... or right after... it won't matter, say `$this->entrypointLookup->reset()`. This tells Encore to *forget* that it rendered anything and forces it to return the same array of CSS files on each call.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 19
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 22 - 49
    protected function execute(InputInterface $input, OutputInterface $output)
    {
// ... lines 52 - 56
        foreach ($authors as $author) {
// ... lines 58 - 62
            if (count($articles) === 0) {
                continue;
            }

            $this->entrypointLookup->reset();
// ... lines 68 - 83
        }
// ... lines 85 - 87
    }
}
```

This *should* make our PDF wonderful. Run the command one more time:

```
php bin/console app:author-weekly-report:send
```

Fly over to Mailtrap and... I'll refresh. Ok, two emails - let's check the second: that's the one what was broken before. The attachment... looks *perfect*.

Next, I like to keep my email logic close together and organized - it helps me to keep emails consistent and, honestly, remember what emails we're sending. Let's refactor the emails into a service... and eventually, use that to write a unit test.

Chapter 18: Organizing Emails Logic into a Service

We're sending two emails: one from a command and the other from `src/Controller/SecurityController.php`. The logic for creating and sending these emails is fairly simple. But even still, I prefer to put all my email logic into one or more *services*. The *real* reason for this is that I like to have all my emails in one spot. That helps me remember *which* emails we're sending and what they contain. After all, emails are a *strange* part of your site: you send a lot of them... but rarely or *never* see them! Like, how often do you do a "password reset" on your own site to check out what that content looks like? Keeping things in one spot... at least helps with this.

Creating a Mailer Service

So what we're going to do is, in the `Service/` directory, create a new class called `FileThatWillSendAllTheEmails`... ah, or, maybe just `Mailer`... it's shorter.

src/Service/Mailer.php

```
// ... lines 1 - 2
namespace App\Service;
// ... lines 4 - 6
class Mailer
{
// ... lines 9 - 14
}
```

The idea is that this class will have one method for *each* email that our app sends. Now, if your app sends a *lot* of emails, instead of having just *one* `Mailer` class, you could instead create a `Mailer/` directory with a bunch of service classes inside - like one per email. In both cases, you're either organizing your email logic into a single service or multiple services in one directory.

Start by adding an `__construct()` method. The *one* service that we *know* we're going to need is `MailerInterface $mailer`... because we're going to send emails. I'll hit Alt + Enter and go to "Initialize fields" to create that property and set it.

```php
// src/Service/Mailer.php
// ... lines 1 - 2
namespace App\Service;

use Symfony\Component\Mailer\MailerInterface;

class Mailer
{
    private $mailer;

    public function __construct(MailerInterface $mailer)
    {
        $this->mailer = $mailer;
    }
}
```

Ok, let's start with the registration email inside of `SecurityController`. Ok... to send this email, the only info we need is the `User` object. Create a new public function `sendWelcomeMessage()` with a `User $user` argument.

```php
// src/Service/Mailer.php
// ... lines 1 - 4
use App\Entity\User;
// ... lines 6 - 12
class Mailer
{
// ... lines 15 - 27
    public function sendWelcomeMessage(User $user)
    {
// ... lines 30 - 40
    }
// ... lines 42 - 63
}
```

Then, grab the logic from the controller... everything from `$email =` to the sending part... and paste that here. It looks like this class is missing a few `use` statements... so I'll re-type the "L" on `TemplatedEmail` and hit tab, then re-type the `S` on `NamedAddress` and hit tab once more... to add those `use` statements to the top of this file. Then change `$mailer` to `$this->mailer`.

Tip

In Symfony 4.4 and higher, use `new Address()` - it works the same way as the old `NamedAddress`.

src/Service/Mailer.php

```php
// ... lines 1 - 6
use Symfony\Bridge\Twig\Mime\TemplatedEmail;
// ... line 8
use Symfony\Component\Mime\NamedAddress;
// ... lines 10 - 12
class Mailer
{
// ... lines 15 - 27
    public function sendWelcomeMessage(User $user)
    {
        $email = (new TemplatedEmail())
            ->from(new NamedAddress('alienmailcarrier@example.com', 'The Space Bar'))
            ->to(new NamedAddress($user->getEmail(), $user->getFirstName()))
            ->subject('Welcome to the Space Bar!')
            ->htmlTemplate('email/welcome.html.twig')
            ->context([
                // You can pass whatever data you want
                //'user' => $user,
            ]);

        $this->mailer->send($email);
    }
// ... lines 42 - 63
}
```

I love it! This will simplify life dramatically in `SecurityController`. Delete all the logic and then above... replace the `MailerInterface` argument with our shiny new `Mailer` class.

```
src/Controller/SecurityController.php
// ... lines 1 - 8
use App\Service\Mailer;
// ... lines 10 - 20
class SecurityController extends AbstractController
{
// ... lines 23 - 50
    public function register(Mailer $mailer, Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guardHandler, LoginFormAuthenticator $formAuthenticator)
    {
// ... lines 53 - 89
    }
}
```

Below, it's as simple as `$mailer->sendWelcomeMessage($user)`.

```
src/Controller/SecurityController.php
// ... lines 1 - 8
use App\Service\Mailer;
// ... lines 10 - 20
class SecurityController extends AbstractController
{
// ... lines 23 - 50
    public function register(Mailer $mailer, Request $request, UserPasswordEncoderInterface $passwordEncoder, GuardAuthenticatorHandler $guardHandler, LoginFormAuthenticator $formAuthenticator)
    {
// ... lines 53 - 55
        if ($form->isSubmitted() && $form->isValid()) {
// ... lines 57 - 74
            $em->flush();

            $mailer->sendWelcomeMessage($user);
// ... lines 78 - 84
        }
// ... lines 86 - 89
    }
}
```

That looks really nice! Our controller is now more readable.

Let's repeat the same thing for our weekly report email. In this case, the two things we need are the `$author` that we're going to send to - which is a `User` object - and the array of articles. Ok, over in our new `Mailer` class, add a public function `sendAuthorWeeklyReportMessage()` with a `User` object argument called `$author` and an array of `Article` objects.

```
src/Service/Mailer.php
// ... lines 1 - 12
class Mailer
{
// ... lines 15 - 42
    public function sendAuthorWeeklyReportMessage(User $author, array $articles)
    {
// ... lines 45 - 62
    }
}
```

Time to steal some code! Back in the command, copy *everything* related to sending the email... which in this case includes the entrypoint reset, Twig render, PDF code *and* the *actual* email logic. Paste that into `Mailer`.

```
src/Service/Mailer.php
// ... lines 1 - 12
class Mailer
{
// ... lines 15 - 42
    public function sendAuthorWeeklyReportMessage(User $author, array $articles)
    {
        $this->entrypointLookup->reset();
        $html = $this->twig->render('email/author-weekly-report-pdf.html.twig', [
            'articles' => $articles,
        ]);
        $pdf = $this->pdf->getOutputFromHtml($html);

        $email = (new TemplatedEmail())
            ->from(new NamedAddress('alienmailcarrier@example.com', 'The Space Bar'))
            ->to(new NamedAddress($author->getEmail(), $author->getFirstName()))
            ->subject('Your weekly report on the Space Bar!')
            ->htmlTemplate('email/author-weekly-report.html.twig')
            ->context([
                'author' => $author,
                'articles' => $articles,
            ])
            ->attach($pdf, sprintf('weekly-report-%s.pdf', date('Y-m-d')));

        $this->mailer->send($email);
    }
}
```

This time, we need to inject a few more services - for `entrypointLookup`, `twig` and `pdf`. Let's add those on top: `Environment $twig`, `Pdf $pdf` and `EntrypointLookupInterface $entrypointLookup`. I'll do my Alt + Enter shortcut and go to "Initialize fields" to create those three properties and set them.

```php
// src/Service/Mailer.php
// ... lines 1 - 5
use Knp\Snappy\Pdf;
// ... lines 7 - 9
use Symfony\WebpackEncoreBundle\Asset\EntrypointLookupInterface;
use Twig\Environment;
// ... line 12
class Mailer
{
// ... line 15
    private $twig;
    private $pdf;
    private $entrypointLookup;

    public function __construct(MailerInterface $mailer, Environment
$twig, Pdf $pdf, EntrypointLookupInterface $entrypointLookup)
    {
        $this->mailer = $mailer;
        $this->twig = $twig;
        $this->pdf = $pdf;
        $this->entrypointLookup = $entrypointLookup;
    }
// ... lines 27 - 63
}
```

Back in the method... oh... that's it! We're already using the properties... and everything looks happy! Oh, and it's minor, but I'm going to move the "entrypoint reset" code *below* the render. This is subtle... but it makes sure that the Encore stuff is reset *after* we render our template. If some *other* part of our app calls this methods and *then* renders its own template, Encore will *now* be ready to do work correctly for them.

```
src/Service/Mailer.php
// ... lines 1 - 12
class Mailer
{
// ... lines 15 - 42
    public function sendAuthorWeeklyReportMessage(User $author, array $articles)
    {
        $html = $this->twig->render('email/author-weekly-report-pdf.html.twig', [
            'articles' => $articles,
        ]);
        $this->entrypointLookup->reset();
// ... lines 49 - 62
    }
}
```

Anyways, let's use this in the command. Delete *all* of this logic and... in the constructor, change the `$mailer` argument to `Mailer $mailer`. Now we get to delete stuff! Take off the `$twig`, `$pdf` and `$entrypointLookup` arguments, clear them from the constructor and remove their properties. If you *really* want to make things squeaky-clean, we now have a bunch of "unused" `use` statements that are totally useless.

```
src/Command/AuthorWeeklyReportSendCommand.php
// ... lines 1 - 6
use App\Service\Mailer;
// ... lines 8 - 14
class AuthorWeeklyReportSendCommand extends Command
{
// ... lines 17 - 20
    private $mailer;

    public function __construct(UserRepository $userRepository, ArticleRepository $articleRepository, Mailer $mailer)
    {
        parent::__construct(null);

        $this->userRepository = $userRepository;
        $this->articleRepository = $articleRepository;
        $this->mailer = $mailer;
    }
// ... lines 31 - 61
}
```

Back down, call the method with `$this->mailer->sendWeeklyReportMessage()` passing `$author` and `$articles`.

```
tests/MailerTest.php
// ... lines 1 - 2
namespace App\Tests;

use PHPUnit\Framework\TestCase;

class MailerTest extends TestCase
{
    public function testSomething()
    {
        $this->assertTrue(true);
    }
}
```

Phew! This *really* simplifies the controller & command... and now I know *exactly* where to look for all email-related code. Let's... just make sure I didn't break anything. Run:

```
php bin/console app:author-weekly-report:send
```

No errors... and in Mailtrap... yep! 2 emails... with an attachment!

Next, sending emails is scary! So let's add some tests. We'll start by adding a unit test and later, an integration test, functional test... and a final exam that will be worth 50% of your grade for the semester. Ok... no final exam - but we will do that other stuff.

Chapter 19: Unit Testing our Emails

Other than code organization, one of the benefits of putting logic into a service is that we can unit test it. Ok, to be *fully* honest, this chapter doesn't have a lot to do with Mailer. Unit tests *pretty* much look the same no matter *what* you're testing. But unit testing is a great practice... and I hate when code does weird things... especially code that sends emails.

make:unit-test

Let's use MakerBundle to bootstrap a test for us. At your terminal, run:

```
php bin/console make:unit-test
```

Answer `MailerTest`. This generates a *super* simple unit test file: `tests/MailerTest.php`.

tests/MailerTest.php

```php
// ... lines 1 - 2
namespace App\Tests;

use PHPUnit\Framework\TestCase;

class MailerTest extends TestCase
{
    public function testSomething()
    {
        $this->assertTrue(true);
    }
}
```

The idea is that this will test the `Mailer` class, which lives in the `Service/` directory. Inside `tests/`, create a new `Service/` directory to match that and move `MailerTest` inside. You typically want your test directory structure to match your `src/` structure. Inside the file, don't forget to add `\Service` to the namespace to match the new location.

```
tests/Service/MailerTest.php
// ... lines 1 - 2
namespace App\Tests\Service;
// ... lines 4 - 6
class MailerTest extends TestCase
{
// ... lines 9 - 12
}
```

Running the Tests

Ok! Our test asserts that true is true! I'm not so easily convinced... we better run PHPUnit to be sure. At your terminal, run it with:

```
php bin/phpunit
```

This script is a small wrapper around PHPUnit... and it will *install* PHPUnit the first time you run it. Then... it passes!

Oh! But it *did* print out a deprecation notice. One of the superpowers of this wrapper around PHPUnit - called the phpunit-bridge - is that it prints out warnings about any deprecated code that the code in your tests hit. This is a great tool when you're getting ready to upgrade your app to the next major Symfony version. But more on that in a future tutorial. We'll just ignore these.

Go Deeper!

If PHPUnit is new for you - or you just want to go deeper - check out our dedicated PHPUnit Tutorial.

Writing the Unit Test

Let's get to work! So... what *are* we going to test? Well, we probably want to test that the mail was actually *sent*... and maybe we'll assert a few things about the `Email` object itself. Unit tests always start the same way: by instantiating the class you want to test.

Back in `MailerTest`, rename the method to `testSendWelcomeMessage()`.

```
tests/Service/MailerTest.php
// ... lines 1 - 12
class MailerTest extends TestCase
{
    public function testSendWelcomeMessage()
    {
// ... lines 17 - 30
    }
}
```

Then add `$mailer = new Mailer()`. For this to work, we need to pass the 4 dependencies: objects of the types `MailerInterface`, `Twig`, `Pdf` and `EntrypointLookupInterface`. In a unit test, instead of using *real* objects that really *do* send emails... or render Twig templates, we use mocks.

For the first, say `$symfonyMailer = this->createMock()`... and because the first argument needs to be an instance of `MailerInterface`, that's what we'll mock: `MailerInterface::class`.

```
tests/Service/MailerTest.php
// ... lines 1 - 8
use Symfony\Component\Mailer\MailerInterface;
// ... lines 10 - 12
class MailerTest extends TestCase
{
    public function testSendWelcomeMessage()
    {
        $symfonyMailer = $this->createMock(MailerInterface::class);
// ... lines 18 - 30
    }
}
```

To make sure we don't forget to actually *send* the email, we can add an assertion to this mock: we can tell PHPUnit that the `send` method *must* be called exactly one time. Do that with `$symfonyMailer->expects($this->once())` that the `->method('send')` is called.

```
tests/Service/MailerTest.php
// ... lines 1 - 8
use Symfony\Component\Mailer\MailerInterface;
// ... lines 10 - 12
class MailerTest extends TestCase
{
    public function testSendWelcomeMessage()
    {
        $symfonyMailer = $this->createMock(MailerInterface::class);
        $symfonyMailer->expects($this->once())
            ->method('send');
// ... lines 20 - 30
    }
}
```

Let's create the 3 other mocks: `$pdf = this->createMock(Pdf::class)`... and the other two are for `Environment` and `EntrypointLookupInterface`: `$twig = $this->createMock(Environment::class)` and `$entrypointLookup = $this->createMock(EntrypointLookupInterface::class)`.

```
tests/Service/MailerTest.php
// ... lines 1 - 6
use Knp\Snappy\Pdf;
// ... line 8
use Symfony\Component\Mailer\MailerInterface;
use Symfony\WebpackEncoreBundle\Asset\EntrypointLookupInterface;
use Twig\Environment;
// ... line 12
class MailerTest extends TestCase
{
    public function testSendWelcomeMessage()
    {
        $symfonyMailer = $this->createMock(MailerInterface::class);
        $symfonyMailer->expects($this->once())
            ->method('send');

        $pdf = $this->createMock(Pdf::class);
        $twig = $this->createMock(Environment::class);
        $entrypointLookup = $this-
>createMock(EntrypointLookupInterface::class);
// ... lines 24 - 30
    }
}
```

These three objects aren't even used in this method... so we don't need to add any assertions to them or configure any behavior. Finish the `new Mailer()` line by passing `$symfonyMailer`, `$twig`, `$pdf` and `$entrypointLookup`. Then, call the method: `$mailer->sendWelcomeMessage()`. Oh, to do *this*, we need a `User` object.

tests/Service/MailerTest.php

```php
// ... lines 1 - 5
use App\Service\Mailer;
use Knp\Snappy\Pdf;
use PHPUnit\Framework\TestCase;
use Symfony\Component\Mailer\MailerInterface;
use Symfony\WebpackEncoreBundle\Asset\EntrypointLookupInterface;
use Twig\Environment;

class MailerTest extends TestCase
{
    public function testSendWelcomeMessage()
    {
        $symfonyMailer = $this->createMock(MailerInterface::class);
        $symfonyMailer->expects($this->once())
            ->method('send');

        $pdf = $this->createMock(Pdf::class);
        $twig = $this->createMock(Environment::class);
        $entrypointLookup = $this->createMock(EntrypointLookupInterface::class);
// ... lines 24 - 28
        $mailer = new Mailer($symfonyMailer, $twig, $pdf, $entrypointLookup);
        $mailer->sendWelcomeMessage($user);
    }
}
```

Should we mock the `User` object? We could, but as a general rule, I like to mock services but manually instantiate simple "data" objects, like Doctrine entities. The reason is that these classes don't have dependencies and it's usually dead-simple to put whatever data you need on them. Basically, it's easier to create the *real* object, than create a mock.

Start with `$user = new User()`. And... let's see... the only information that we use from `User` is the email and first name. For `$user->setFirstName()`, let's pass the name of my brave co-author for this tutorial: `Victor`! And for `$user->setEmail()`, him again `victor@symfonycasts.com`. Give this `$user` variable to the `sendWelcomeMessage()` method.

```
tests/Service/MailerTest.php
// ... lines 1 - 4
use App\Entity\User;
use App\Service\Mailer;
use Knp\Snappy\Pdf;
use PHPUnit\Framework\TestCase;
use Symfony\Component\Mailer\MailerInterface;
use Symfony\WebpackEncoreBundle\Asset\EntrypointLookupInterface;
use Twig\Environment;

class MailerTest extends TestCase
{
    public function testSendWelcomeMessage()
    {
        $symfonyMailer = $this->createMock(MailerInterface::class);
        $symfonyMailer->expects($this->once())
            ->method('send');

        $pdf = $this->createMock(Pdf::class);
        $twig = $this->createMock(Environment::class);
        $entrypointLookup = $this->createMock(EntrypointLookupInterface::class);

        $user = new User();
        $user->setFirstName('Victor');
        $user->setEmail('victor@symfonycasts.com');

        $mailer = new Mailer($symfonyMailer, $twig, $pdf, $entrypointLookup);
        $mailer->sendWelcomeMessage($user);
    }
}
```

By the way, if you're enjoying this tutorial, you can thank Victor personally by emailing him photos of your cat *or* by sending tuna *directly* to his cat Ponka.

And... done! We're not asserting anything down *here*... but we *do* have one built-in assert above: our test will fail unless the `send()` method is called exactly once.

Let's try this! Fly over to your terminal, I'll clear my screen, then run:

```
php bin/phpunit
```

It passes! The power!

Asserting Info on the Email

The tricky thing is that the majority of this method is about creating the `Email`... and we're not testing what *that* object looks like at all. And... maybe we don't need to? I tend to unit test logic that scares me and manually test other things - like the wording inside an email. But let's *at least* assert a few basic things.

How? An easy way is to return the email from each method: `return $email` and then advertise that this method returns a `TemplatedEmail`. I'll do the same for the other method: `return $email` and add the `TemplatedEmail` return type.

```
src/Service/Mailer.php
// ... lines 1 - 12
class Mailer
{
// ... lines 15 - 27
    public function sendWelcomeMessage(User $user): TemplatedEmail
    {
// ... lines 30 - 41
        return $email;
    }
// ... line 44
    public function sendAuthorWeeklyReportMessage(User $author, array $articles): TemplatedEmail
    {
// ... lines 47 - 65
        return $email;
    }
}
```

You don't *have* to do this, but it'll make our unit test more useful and keep it simple. *Now* we can say `$email = $mailer->sendWelcomeMessage()` and we can check pretty much *anything* on that email.

I'll paste in some asserts:

```
tests/Service/MailerTest.php
// ... lines 1 - 13
class MailerTest extends TestCase
{
    public function testSendWelcomeMessage()
    {
// ... lines 18 - 29
        $mailer = new Mailer($symfonyMailer, $twig, $pdf, $entrypointLookup);
        $email = $mailer->sendWelcomeMessage($user);

        $this->assertSame('Welcome to the Space Bar!', $email->getSubject());
        $this->assertCount(1, $email->getTo());
        /** @var NamedAddress[] $namedAddresses */
        $namedAddresses = $email->getTo();
        $this->assertInstanceOf(NamedAddress::class, $namedAddresses[0]);
        $this->assertSame('Victor', $namedAddresses[0]->getName());
        $this->assertSame('victor@symfonycasts.com', $namedAddresses[0]->getAddress());
    }
}
```

Tip

In Symfony 4.4 and higher, use `new Address()` - it works the same way as the `NamedAddress` we use here.

These check the subject, that the email is sent to exactly one person *and* checks to make sure that the "to" has the right info.

Let's give this a try! Move over and run:

```
php bin/phpunit
```

All green! Next, let's do this same thing for the author weekly report email. Actually... the "email" part of this method is, once again, *pretty* simple. The *complex* part is the PDF-generation logic. Want to test to make sure the template *actually* renders correctly and the PDF is *truly* created? We can't do that with a pure unit test... but we *can* with an integration test. That's next.

Chapter 20: Integration Testing Emails

I *also* want to test the method that sends the weekly update email. But because the *real* complexity of this method is centered around generating the PDF, instead of a unit test, let's write an *integration* test.

In `MailerTest`, add a second method: `testIntegrationSendAuthorWeeklyReportMessage()`.

```
tests/Service/MailerTest.php
// ... lines 1 - 14
class MailerTest extends TestCase
{
// ... lines 17 - 42
    public function testIntegrationSendAuthorWeeklyReportMessage()
    {
// ... lines 45 - 58
    }
}
```

Let's start the same way as the first method: copy *all* of its code except for the asserts, paste them down here and change the method to `sendAuthorWeeklyReportMessage()`.

```
tests/Service/MailerTest.php
// ... lines 1 - 14
class MailerTest extends TestCase
{
// ... lines 17 - 42
    public function testIntegrationSendAuthorWeeklyReportMessage()
    {
        $symfonyMailer = $this->createMock(MailerInterface::class);
        $symfonyMailer->expects($this->once())
            ->method('send');

        $pdf = $this->createMock(Pdf::class);
        $twig = $this->createMock(Environment::class);
        $entrypointLookup = $this->createMock(EntrypointLookupInterface::class);

        $user = new User();
        $user->setFirstName('Victor');
        $user->setEmail('victor@symfonycasts.com');

        $mailer = new Mailer($symfonyMailer, $twig, $pdf, $entrypointLookup);
        $email = $mailer->sendWelcomeMessage($user);
    }
}
```

This needs a `User` object... but it also needs an array of articles. Let's create one: `$article = new Article()`. These articles are passed to the template where we print their title. So let's at least populate that property: `$article->setTitle()`:

> *“Black Holes: Ultimate Party Pooper”*

```
tests/Service/MailerTest.php
// ... lines 1 - 4
use App\Entity\Article;
// ... lines 6 - 14
class MailerTest extends TestCase
{
// ... lines 17 - 42
    public function testIntegrationSendAuthorWeeklyReportMessage()
    {
// ... lines 45 - 52
        $user = new User();
        $user->setFirstName('Victor');
        $user->setEmail('victor@symfonycasts.com');
        $article = new Article();
        $article->setTitle('Black Holes: Ultimate Party Pooper');
// ... lines 58 - 60
    }
}
```

Use this for the 2nd argument of `sendAuthorWeeklyReportMessage()`: an array with just this inside.

```
tests/Service/MailerTest.php
// ... lines 1 - 14
class MailerTest extends TestCase
{
// ... lines 17 - 42
    public function testIntegrationSendAuthorWeeklyReportMessage()
    {
// ... lines 45 - 52
        $user = new User();
        $user->setFirstName('Victor');
        $user->setEmail('victor@symfonycasts.com');
        $article = new Article();
        $article->setTitle('Black Holes: Ultimate Party Pooper');

        $mailer = new Mailer($symfonyMailer, $twig, $pdf, $entrypointLookup);
        $email = $mailer->sendAuthorWeeklyReportMessage($user, [$article]);
    }
}
```

Unit Versus Integration Test

It's time to think strategically about our mocks. Right now, *every* dependency is mocked, which means it's a *pure* unit test. If we kept doing this, we could probably make sure that whatever `render()` returns is passed to the PDF function... and even assert that whatever *that* returns is passed to the `attach()` method. It's not bad, but because the *logic* in this method isn't terribly complex, its usefulness is limited.

What *really* scares me is the PDF generation: does my Twig template render correctly? Does the PDF generation process work... and do I *really* get back PDF content? To test this, instead of mocking `$twig` and `$pdf`, we could use the *real* objects. That would make this an *integration* test. These are often more useful than unit tests... but are also much slower to run, and it will mean that I really *do* need to have `wkhtmltopdf` installed on this machine, otherwise my tests will fail. Tradeoffs!

So here's the plan: use the *real* `$twig` and `$pdf` objects but *keep* mocking `$symfonyMailer` and `$entrypointLookup`... because I don't *really* want to send emails... and the `$entrypointLookup` doesn't matter unless I want to test that it *does* reset things correctly between rendering 2 PDFs.

Become an Integration Test!

To make this test *able* to use real objects, we need to change `extends` from `TestCase` to `KernelTestCase`.

tests/Service/MailerTest.php
```
// ... lines 1 - 9
use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;
// ... lines 11 - 15
class MailerTest extends KernelTestCase
{
// ... lines 18 - 62
}
```

That class extends the *normal* `TestCase` but gives us the ability to boot Symfony's service container in the background. Specifically, it gives us the ability, down in the method, to say: `self::bootKernel()`.

```
tests/Service/MailerTest.php
// ... lines 1 - 9
use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;
// ... lines 11 - 15
class MailerTest extends KernelTestCase
{
// ... lines 18 - 43
    public function testIntegrationSendAuthorWeeklyReportMessage()
    {
        self::bootKernel();
        $symfonyMailer = $this->createMock(MailerInterface::class);
// ... lines 48 - 61
    }
}
```

That will give us the ability to fetch *real* service objects and use them.

Fetching out Services

So we'll leave `$symfonyMailer` mocked, leave the `$entrypointLookup` mocked, but for the `Pdf`, get the *real* `Pdf` service. How? In the test environment, we can fetch things out of the container using the same type-hints as normal. So, `$pdf = self::$container` - `bootKernel()` set that property - `->get()` passing this `Pdf::class`. Do the same for Twig: `self::$container->get(Environment::class)`.

Tip

Starting in Symfony 5.3, instead of `self::$container`, use `static::getContainer()` to get the container from inside a test. Also, calling `bootKernel()` is no longer needed.

```
tests/Service/MailerTest.php
// ... lines 1 - 9
use Symfony\Bundle\FrameworkBundle\Test\KernelTestCase;
// ... lines 11 - 15
class MailerTest extends KernelTestCase
{
// ... lines 18 - 43
    public function testIntegrationSendAuthorWeeklyReportMessage()
    {
        self::bootKernel();
// ... lines 47 - 49
        $pdf = self::$container->get(Pdf::class);
        $twig = self::$container->get(Environment::class);
// ... lines 52 - 61
    }
}
```

I love that! Again, the *downside* is that you really *do* need to have `wkhtmltopdf` installed correctly *anywhere* you run your tests. That's the *cost* of doing this.

Before we try it, at the bottom, we don't have any asserts yet. Let's add at least one: `$this->assertCount()` that 1 is the count of `$email->getAttachments()`.

```
tests/Service/MailerTest.php
// ... lines 1 - 15
class MailerTest extends KernelTestCase
{
// ... lines 18 - 43
    public function testIntegrationSendAuthorWeeklyReportMessage()
    {
// ... lines 46 - 60
        $email = $mailer->sendAuthorWeeklyReportMessage($user,
[$article]);
        $this->assertCount(1, $email->getAttachments());
    }
}
```

We *could* go further and look closer at the attachment... maybe make sure that it looks like it's in a PDF format... but this is a good start.

Now let's try this. Find your terminal and run our normal:

```
php bin/phpunit
```

It *is* slower this time... and then.. ah! What just happened? Two things. First, because this booted up a *lot* more code, we're seeing a *ton* of deprecation warnings. These are annoying... but we can ignore them.

Caching Driver in the test Environment

The *second* thing is that... the test failed! But... weird - not how I expected: something about APCu is not enabled. Huh? Why is it suddenly trying to use APCu?

The cause of this is specific to our app... but it's an interesting situation. Open up `config/packages/cache.yaml`.

config/packages/cache.yaml

```yaml
framework:
    cache:
// ... lines 3 - 14
        app: '%cache_adapter%'
// ... lines 16 - 21
```

See this `app` key? This is where you can tell Symfony *where* it should store things that need to be added to cache at runtime - like the filesystem, redis or APCu. In an earlier tutorial, we set this to a parameter that we invented: `%cache_adapter%`.

This allows us to do something cool. Open `config/services.yaml`.

config/services.yaml

```yaml
// ... lines 1 - 5
parameters:
    cache_adapter: cache.adapter.apcu
// ... lines 8 - 53
```

Here, we set `cache_adapter` to `cache.adapter.apcu`: we told Symfony to store cache in APCu. And... apparently, I don't have that extension installed on my local machine.

Ok... fine... but then... how the heck is the website working? Shouldn't we be getting this error everywhere? Yep... except that we *override* this value in `services_dev.yaml` - a file that is *only* loaded in the `dev` environment. Here we tell it to use `cache.adapter.filesystem`.

config/services_dev.yaml

```yaml
parameters:
    cache_adapter: 'cache.adapter.filesystem'
```

This is great! It means that we don't need any special extension for the cache system while developing... but on production, we use the superior APCu.

The problem *now* is that, when we run our tests, those are run in the `test` environment... and since the `test` environment doesn't load `services_dev.yaml`, it's using the default APCu adapter! By the way, there *is* a `services_test.yaml` file... but it has nothing in it. In fact, you can delete this: it's for a feature that's not needed anymore.

So, honestly... I *should* have set this all up better. And now, I will. Change the default cache adapter to `cache.adapter.filesystem`.

```
config/services.yaml
// ... lines 1 - 5
parameters:
    cache_adapter: cache.adapter.filesystem
// ... lines 8 - 53
```

Then, *only* in the `prod` environment, let's change this to `apcu`. To do that, rename `services_dev.yaml` to `services_prod.yaml`... and change the parameter inside to `cache.adapter.apcu`.

```
config/services_prod.yaml
parameters:
    cache_adapter: 'cache.adapter.apcu'
```

Now the `test` environment *should* use the filesystem. Let's try it!

```
php bin/phpunit
```

And... if you ignore the deprecations... it worked! It actually generated the PDF inside the test! To *totally* prove it, real quick, in the test, `var_dump($email->getAttachments())`... and run the test again:

```
php bin/phpunit
```

Yea! It's *so* ugly. The attachment is some `DataPart` object and you can see the crazy PDF content inside. Go take off that dump.

Ok, the *last* type of test is a *functional* test. And this is where things get more interesting... especially in relation to Mailer. If we want to make a functional test for the registration form... do we expect our test to send a *real* email? Or should we disable email delivery somehow while testing? And, in both cases, is it possible to submit the registration form in a functional test and then *assert* that an email *was* in fact sent? Ooo. This is good stuff!

Chapter 21: Functional Testing with Emails

When we originally added our Mailtrap config... I was a bit lazy. I put the value into `.env`. But because that file is *committed*... we *really* shouldn't put any sensitive values into it. Well, you could argue that Mailtrap credentials aren't *that* sensitive, but let's fix this properly. Copy the `MAILER_DSN` and open `.env.local`.

If you don't have a `.env.local` file yet, just create it. I already have one so that I can customize my local database config. The values in this file override the ones in `.env`. And because *this* file is ignored by `.gitignore`, these values won't be committed.

Back in `.env`, let's set `MAILER_DSN` back to the original value, which was `smtp://localhost`.

```
.env
// ... lines 1 - 37
###> symfony/mailer ###
MAILER_DSN=smtp://localhost
###

// ... lines 41 - 46
```

And yes, this *does* mean that when a developer clones the project, unless they customize `MAILER_DSN` in their *own* `.env.local` file, they'll get an error if they try to register... or do anything that sends an email. We'll talk more about that in a few minutes.

Creating a Functional Test

Back to my *real* goal: writing a functional test for the registration page. Because a successful registration causes an email to be sent... I'm curious how that will work. Will an email *actually* be sent to Mailtrap? Do we want that?

To create the test, be lazy and run:

```
php bin/console make:functional-test
```

And... we immediately get an error: we're missing some packages. I'll copy the `composer require browser-kit` part. Panther isn't *technically* needed to write functional tests... and this error message is fixed in a newer version of this bundle. But, Panther *is* an awesome way to write functional tests that rely on JavaScript.

Anyways, run

```
composer require browser-kit --dev
```

... and we'll wait for that to install. Once it finishes, I'll clear the screen and try `make:functional-test` again:

```
php bin/console make:functional-test
```

Access granted! I want to test `SecurityController` - specifically the `SecurityController::register()` method. I'll follow the same convention we used for the unit test: call the class `SecurityControllerTest`.

Done! This creates a simple functional test class directly inside of `tests/`.

```
tests/SecurityControllerTest.php
// ... lines 1 - 2
namespace App\Tests;

use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;

class SecurityControllerTest extends WebTestCase
{
    public function testSomething()
    {
        $client = static::createClient();
        $crawler = $client->request('GET', '/');

        $this->assertResponseIsSuccessful();
        $this->assertSelectorTextContains('h1', 'Hello World');
    }
}
```

We don't *have* to, but to make this match the `src/Controller` directory structure, create a new `Controller/` folder inside of `tests/`... and move the test file there. Don't forget to add `\Controller` to the end of its namespace.

```
tests/Controller/SecurityControllerTest.php
// ... lines 1 - 2
namespace App\Tests\Controller;
// ... lines 4 - 6
class SecurityControllerTest extends WebTestCase
{
// ... lines 9 - 16
}
```

And, again, to stay somewhat conventional, let's rename the method to `testRegister()`.

```
tests/Controller/SecurityControllerTest.php
// ... lines 1 - 6
class SecurityControllerTest extends WebTestCase
{
    public function testRegister()
    {
// ... lines 11 - 24
    }
}
```

Writing the Registration Functional Test

We won't go *too* deep into the details of how to write functional tests, but it's a *pretty* simple idea. First, we create a `$client` object - which is almost like a "browser": it helps us make requests to our app. In this case, we want to make a `GET` request to `/register` to load the form.

```
tests/Controller/SecurityControllerTest.php
// ... lines 1 - 6
class SecurityControllerTest extends WebTestCase
{
    public function testRegister()
    {
        $client = static::createClient();
        $crawler = $client->request('GET', '/register');
// ... lines 13 - 24
    }
}
```

The `assertResponseIsSuccessful()` method is a helper assertion from Symfony that will make sure the response wasn't an error or a redirect.

```
tests/Controller/SecurityControllerTest.php
// ... lines 1 - 6
class SecurityControllerTest extends WebTestCase
{
    public function testRegister()
    {
        $client = static::createClient();
        $crawler = $client->request('GET', '/register');

        $this->assertResponseIsSuccessful();
// ... lines 15 - 24
    }
}
```

Now... I'll remove the `assertSelectorTextContains()`... and paste in the rest of the test.

```
tests/Controller/SecurityControllerTest.php
// ... lines 1 - 6
class SecurityControllerTest extends WebTestCase
{
    public function testRegister()
    {
        $client = static::createClient();
        $crawler = $client->request('GET', '/register');

        $this->assertResponseIsSuccessful();

        $button = $crawler->selectButton('Register');
        $form = $button->form();
        $form['user_registration_form[firstName]']->setValue('Ryan');
        $form['user_registration_form[email]']->setValue(sprintf('foo%s@example.com', rand()));
        $form['user_registration_form[plainPassword]']->setValue('space_rocks');
        $form['user_registration_form[agreeTerms]']->tick();
        $client->submit($form);

        $this->assertResponseRedirects();
    }
}
```

Let's see: this goes to `/register`, finds the `Register` button by its text, and then fills out all the form fields. These funny-looking values are literally the *name* attributes of each element if you looked at the source HTML. After submitting the form, we assert that the response is a redirect... which is an easy way to assert that the form submit *was* successful. If there's a validation error, it re-renders *without* redirecting.

We've used the registration form on this site... about 100 times. So we *know* it works... and so this test *should* pass. Whenever you say that something "should" work in programming... do you ever get the sinking feeling that you're about to eat your words? Ah, I'm sure nothing bad will happen in this case. Let's try it!

At your terminal, run *just* this test with:

```
php bin/phpunit tests/Controller/SecurityControllerTest.php
```

Deprecation notices of course... and... woh! It failed! And dumped some *giant* HTML... which is impossible to read... unless you go *all* the way to the top. Ah!

> *“Failed asserting that the Response is redirected: 500 internal server error.”*

And down in the HTML:

> *“Connection could not be established with host tcp://localhost:25”*

The test Environment Doesn't Read .env.local

Huh. That's coming from sending the email... but why is it trying to connect to `localhost`? Our config in `.env.local` is set up to talk to Mailtrap.

Well... there's a little gotcha about the `.env` system. I mean... it's a feature! When you're in the `test` environment, the `.env.local` file is *not* loaded. In *every* other situation - like the `prod` or the `dev` environments - it *is* loaded. But in `test`, it's *not*. It's madness!

Well, it definitely *is* surprising the first time you see this, but there *is* a good reason for it. In theory, your committed `.env.test` file should contain *all* the configuration needed for the `test` environment to work... on any machine. And so, you actually *don't* want your local values from `.env.local` to override the stuff in `.env.test` - that might *break* how the tests are supposed to behave.

The point is, since the `.env.local` file is not being loaded in our tests, it's using the `.env` settings for `MAILER_DSN`... which is connecting to `localhost`.

How can we fix this? The simplest answer is to copy the `MAILER_DSN` from `.env.local` into `.env.test`. This isn't a *great* solution because `.env.test` is committed... and so we would once again be committing our Mailtrap credentials to the repository. You *can* get around this by creating a `.env.test.local` file - that's a file that's loaded in the `test` environment but *not* committed - but let's just do this for now and see if we can get things working. Later, we'll talk about a better option.

Ok, go tests go!

```
php bin/phpunit tests/Controller/SecurityControllerTest.php
```

This time... it passes! Spin back over and inside Mailtrap... there it is! The test *actually* sent an email! Wait... is that what we want? Let's improve this next by *preventing* emails from our test from *actually* being delivered. Then, we'll talk about how we can add *assertions* to *guarantee* that the right email was sent.

Chapter 22: Email Delivery & Assertions in Tests

We *just* got our registration functional test to pass. But to do it, we had to configure the test environment with our Mailtrap credentials. And that means that each time we run our tests, an email is *actually* being delivered to Mailtrap!

Ok, in reality, because we're using Mailtrap... we're not *really* sending test emails to *real* people. But delivering emails inside our tests is a bummer for a few reasons: it adds a lot of garbage emails to Mailtrap, it slows down our tests *and* it means that we need to worry about configuring *real* Mailtrap credentials *just* to check if our registration test passes.

The truth is, we don't *really* need emails to be sent in the `test` environment. We *do* want the `Email` objects to be created and processed by Mailer... but if at the *last* second Mailer... just... didn't *actually* deliver them... that would be cool! We could *try* to do this by, maybe adding an if statement around `$this->mailer->send()` if we're in the `test` environment... but that would be a pain... and *ugly*.

The Null Transport

Way earlier in the tutorial, I mentioned that the *way* an email is *delivered* is called a "transport". In `.env`, we're using the `smtp` transport to talk to the `localhost` server. In `.env.local`, this is *also* using the `smtp` transport to talk to the Mailtrap server. So far, `smtp` is the *only* transport we've seen.

Well, prepare to be amazed! Introducing the *laziest*, do-nothing... but mysteriously useful transport ever: the `null` transport! When you deliver an email via the `null` transport... your email goes... nowhere.

Hey! That's *exactly* what we want to do in the test environment! Inside `.env.test`, change `MAILER_DSN` to `smtp://null`.

.env.test

```
// ... lines 1 - 5
MAILER_DSN=smtp://null
// ... lines 7 - 9
```

Side note! This syntax *changed* in Symfony 4.4 to `null://default` - where the *start* of the string defines the transport *type*. We'll talk more about transports in a few minutes when we start using SendGrid.

```
.env.test
// ... lines 1 - 5
MAILER_DSN=smtp://null
# in Symfony 4.4 and higher, the syntax is
# MAILER_DSN=null://default
```

Anyways, let's try the test *now*:

```
php bin/phpunit tests/Controller/SecurityControllerTest.php
```

It passes and... yea! There were *no* email sent to Mailtrap. The test *also* ran about twice as fast.

Using the Null Transport by Default?

But wait, there's more! The `null` transport is *perfect* for the test environment. And... it might *also* be a good candidate as the *default* transport.

Hear me out. If a new developer cloned this project, they would *not* have a `.env.local` file. And so, out-of-the-box, mailer would use the `smtp://localhost` setting. What if this developer was really a designer that wanted to work on styling the registration process. Well... surprise! The *moment* they submit the form successfully, they'll be congratulated with a lovely 500 error. And they'll be off to find *you* to figure out how to fix it. That's no good for anyone.

That's why using the `null` transport in `.env` might be a *perfect* default. Then, if someone *actually* wants to *test* how the emails look, *then* they can take some time to configure their `.env.local` file to use Mailtrap.

Let's do this: change `MAILER_DSN` to `smtp://null`. Use `null://default` on Symfony 4.4 or higher.

```
.env
// ... lines 1 - 37
###> symfony/mailer ###
MAILER_DSN=smtp://null
# in Symfony 4.4 and higher, the syntax is
# MAILER_DSN=null://default
###

// ... lines 43 - 48
```

Over in `.env.test`, we don't need to override anything. So, remove `MAILER_DSN`.

Asserting Emails were Sent

We can now use the site *and* run our tests without needing to manually configure mailer. Cool! But we can still make our functional test a *little* bit more fun.

In `SecurityControllerTest`, we *are* testing that the registration form works. But we are *not* asserting that an email *was* in fact sent or... that the email has the right details.

And, while that might not be a huge deal, we *can* add these types of assertions. Well, actually *I* can't add them... because *this* project uses Symfony 4.3. Symfony 4.4 adds a number of new features that make this a *pleasure*.

Google for "Symfony 4.4 mailer testing" to find a blog post about this fancy new stuff. It's... just... awesome. The setup is the same, but after each request, you can choose from a *bunch* of assertions to check that the correct number of emails were sent, that it was sent to the right person, the subject... anything!

In our test class, *after* submitting the form, I'll paste in some assertions that I will use... once I upgrade this app to Symfony 4.4. This checks that one email was sent and then *fetches* the `Email` object itself, which you can then use to make sure *any* part of it is correct.

I'll comment these out for now.

```
tests/Controller/SecurityControllerTest.php
// ... lines 1 - 6
class SecurityControllerTest extends WebTestCase
{
    public function testRegister()
    {
// ... lines 11 - 23
        $this->assertResponseRedirects();

        /* Symfony 4.4:
        $this->assertEmailCount(1);
        $email = $this->getMailerMessage(0);
        $this->assertEmailHeaderSame($email, 'To', 'fabien@symfony.com');
        */
    }
}
```

Next, it's time to send some *real* emails people! It's time to get ready for production! Let's register with a cloud email sender and get it working in our app. We're also going to learn more about Mailer's "transport" system.

Chapter 23: SendGrid & All About Transports

In `.env`, we're using the `null` transport. In `.env.local`, we're overriding that to send to Mailtrap. This is great for development, but it's time for our app to grow up, get a job, and join the real world. It's time for our app to send *real* emails through a *real* email server.

To do that, I recommend using a cloud-based email service... and Symfony Mailer can send to *any* service that supports the SMTP protocol... which is all of them. We did this for Mailtrap using the `{username}:{password}@{server}:{port}` syntax.

But to make life *even* nicer, Mailer has *special* support for the most common email services, like SendGrid, Postmark, Mailgun, Amazon SES and a few others. Let's use SendGrid.

Before we even *create* an account on SendGrid, we can jump in and start configuring it. In `.env.local`, comment-out the Mailtrap `MAILER_DSN` and replace it with `MAILER_DSN=smtp://sendgrid`. In Symfony 4.4, the syntax changed to `sendgrid://default`.

```
#MAILER_DSN=smtp://USERNAME:PASSWORD@smtp.mailtrap.io:2525
MAILER_DSN=smtp://sendgrid
# Symfony 4.4+ syntax
#MAILER_DSN=sendgrid://default
```

All About Transports

So far, we've seen two *transports* - two *ways* of *delivering* emails: the `smtp` transport and the `null` transport. Symfony *also* has a `sendgrid` transport, as well as a `mailgun` transport `amazonses` transport and many others.

In Symfony 4.3, you choose *which* transport you want by saying `smtp://` and then the name of one of those transports, like `null` or `sendgrid`. In Symfony 4.4 and higher, this is different. The syntax is the *transport* name, like `null` or `sendgrid` *then* `://` and whatever other options that transport needs. The word `default` is a dummy placeholder that's used when you don't need to configure a "server", like for the `null` transport or for `sendgrid`, because that transport already knows internally what the address is to the SendGrid servers.

Anyways, whether you're in Symfony 4.3 with the old syntax or Symfony 4.4 with the new one, *this* is how you say: "I want to deliver emails via the SendGrid transport".

At this point, some of you might be *screaming*

> *“Wait! That can't possibly be all the config we need to send emails!”*

And you're 1000% percent correct. This doesn't contain any SendGrid username, or API key. Heck, we haven't even created a SendGrid account yet! All true, all true. But let's... try it anyways. Because, Symfony is going to guide us through the process. How nice!

Let Symfony Guide You to Configure the Transport

Head over to the browser and refresh. Woh! An immediate error:

> *“Unable to send emails via Sendgrid as the bridge is not installed.”*

This is *another* example of Symfony making it easy to do something... but *without* bloating our project with stuff we don't need. Now that we *do* want to use Sendgrid, it helps us install the required library. Copy the `composer require` line, spin over to your terminal and paste:

```
composer require symfony/sendgrid-mailer
```

Ooh, this package came with a recipe! Let's see what it did:

```
git status
```

In addition to the normal stuff, this *also* modified our `.env` file. Let's see how:

```
git diff .env
```

Cool! The recipe added a new section to the bottom! Back in our editor, let's see what's going on in `.env`:

```
.env
// ... lines 1 - 48
###> symfony/sendgrid-mailer ###
# SENDGRID_KEY=
# MAILER_DSN=smtp://$SENDGRID_KEY@sendgrid
###
```

Yea... this makes sense. We know that mailer is configured via a `MAILER_DSN` environment variable... and so when we installed the SendGrid mailer package, its recipe added a *suggestion* of how that variable should look in order to work with SendGrid.

SendGrid Symfony 4.4 Config Format

Now, two important notes about this. First, when you install this package in Symfony 4.4, the config added by the recipe will look a bit different: it will add just one line:

```
MAILER_DSN=sendgrid://KEY@default
```

Like we just talked about, this is because Symfony 4.4 changed the config format: the "transport type" is now at the beginning. The `KEY` is a placeholder: we'll replace with a *real* API key in a few minutes. And the `@default` part just tells the SendGrid transport to send the message to whatever the *actual* SendGrid hostname is.... we don't need to worry about configuring that.

A Note about Environment Variables inside Environment Variables

Now, if you look at the config that Symfony 4.3 uses, you'll notice the second important thing: this defines *two* environment variables. Gasp! It defines `SENDGRID_KEY` and *then* `MAILER_DSN`. This... is just a config trick. See how the `MAILER_DSN` value *contains* `$SENDGRID_KEY`? It's *using* that variable: it's environment variables inside environment variables! With this setup, you could commit this `MAILER_DSN` value to `.env` and then *only* need to override `SENDGRID_KEY` in `.env.local`.

This idea - the idea of using environment variables *inside* environment variables *totally* works in Symfony 4.4. But to keep the config a bit simpler, in Symfony 4.4 - you won't see this two-variable system in the recipe. Instead, we'll configure the *entire* `MAILER_DSN` value. After all, it's a pretty short string.

Next... let's actually *do* that configuration! It's time to create a SendGrid account and start using it.

Chapter 24: Production Settings with SendGrid

If we're going to send emails with SendGrid... we... probably need an account! Head to `sendgrid.com` and click to register. I'll create a shiny new `symfonycasts` username, a thought-provoking password, my email and I am hopefully *not* a robot... and if I am... I'm *at least* a self-aware robot. Does that count? And... create account! Oh man! Registration step 2! Let's fill these out and... done!

SendGrid *just* sent us an email to verify my account. I've already got my inbox open and ready. There it is! I'll click to confirm my email and... we're good!

Creating the SendGrid API Key

Back on the SendGrid "guide" page, on a high-level, we need some sort of API key or username & password that we can use to send through our new account. Click "Start" and then "Choose" the SMTP Relay option.

Yea, I know, I know: SendGrid says that the Web API method is recommended. Most Cloud providers give you these two options: send through the traditional SMTP relay *or* use some custom API endpoints that they expose. They recommend the API way because, if you're creating all of your emails by hand, it's probably easier: just POST your subject, to, from, body, etc to an API endpoint and it takes care of creating the email behind-the-scenes. The API probably also has a few extra, SendGrid-specific features *if* you need to do something really custom.

But because Mailer - and really the Mime component - are handling all of the complexity of creating the email *for* us, it's much easier to use the SMTP relay.

Finally, it's time to create an API key that will authenticate us over SMTP. Give the key a name - just so you can recognize what it's for 1 year from now when we've *completely* forgotten. And hit "Create Key".

Check out our beautiful new SendGrid API key. Hmm, actually, down here, it's called a "Password". In reality, this *is* a SendGrid API key - you *could* use it to send emails through their RESTful API. But because SMTP authentication works via a username and password, SendGrid tells us to use `apikey` as the username and this as the password. It also tells us exactly what server and port to use. This is *everything* we need. Copy the password.

Configuring the SMTP Way vs the SendGrid Transport Way

In `.env.local`, we could use all that info to fill in the normal `smtp://username:password@server:port` format. That would *totally* work.

Or, we could use the SendGrid transport to make life easier: just `smtp://` - the long API key - then `@sendgrid`.

```
MAILER_DSN=smtp://API_KEY@sendgrid
```

The `sendgrid` transport is just a small wrapper around the SMTP transport to make life easier: because it *knows* that the `username` is always `apikey`... and that the server is always `smtp.sendgrid.net`, we don't need to fill those in.

In Symfony 4.4, the new syntax will look like this:

```
sendgrid://KEY@default
```

By the way, the SendGrid transport can use SMTP behind the scenes *or* make API requests to the SendGrid API. In fact, most transports are like this. Symfony chooses the *best* one by default - usually smtp - but you could force it to use the API by saying `sendgrid+api://`.

Sending an Email!

Tip

SendGrid now requires that you "authenticate" your from address before you can send any emails. We'll talk more about "sender authentication" in the next chapter, but to send your first email, you will need to do a few extra steps:
1) Follow https://sendgrid.com/docs/ui/sending-email/sender-verification/ to verify a real email address. For development, you can use your personal email.
2) In `src/Service/Mailer.php`, update the `setFrom()` line to use the email you just configured, instead of `alienmailcarrier@example.com`.

Ok team - let's try this! Back in the browser, tell SendGrid that we *have* updated our settings and click "Next".

At this point, unless we've made a mistake, it *should* work: SendGrid is waiting for us to try it. So... let's do that! Back on our site, hit enter on the registration page. This time, because we're going to send a *real* email - yay! - I'll register with a *real* address: `ryan@symfonycasts.com`. Type in a fun password, agree to terms and... go!

No errors!? Ho, ho! Because it *probably* worked. Tell SendGrid to "Verify Integration" - that makes it *look* for the email we just sent.

Our Message is Spammy

While we're waiting... ah! I see a new message in my inbox! And it looks *perfect*. If you don't see anything, double-check your spam folder. Because... the email we sent is actually *super* spammy. Why? See how we're sending from `alienmailer@example.com`? Do we *own* the `example.com` domain? No! And even if we did, we have not *proven* that our SendGrid account is *allowed* to send emails on behalf of that domain. This is *the* biggest mistake you can make when sending emails and we'll talk more about how to fix it in a few minutes.

But first, back on SendGrid... hmm. It didn't see my email? It *definitely* sent. Hit to verify again - sometimes this works quickly... but I've also had to hit this button 3-4 times before. So... keep trying.

Finally, it works. Next, our great new email system... will probably result in pretty much *every* email we send going straight to Spam. Wah, wah. We need to *prove* that we are *allowed* to send from whatever domain our "from" address is set to. Let's tackle "Sender authentication".

Chapter 25: Sender Authentication: SPF & DKIM

Just configuring your app to use a cloud email sender - like SendGrid - isn't enough. That would be too simple! In my Gmail inbox, the message *was* delivered... but I think we got lucky. This email *smells* like spam. The reason is that we're *claiming* that the email is coming *from* `alienmailer@example.com`. We can see that in our `Mailer` class: every email is coming *from* this address.

In a real app, we would replace this with an email address from our *real* domain - like `droid@thespacebar.com`. But that doesn't fix things. The question still remains: how does Gmail know that SendGrid - or really, our *account* on SendGrid - is *authorized* to send emails from this domain? How does it know that we're not some random spammer or phisher that's trying to *trick* users into thinking this email is legitimately from this domain?

To get our emails past spam blockers, we need to add extra config to our domain's DNS that *proves* our SendGrid account *is* authorized to send emails from `example.com`... or whatever *your* domain actually is.

This is both a simple thing to do... and maybe confusing? Fortunately, every email provider will guide you through the process and... I'll do my best to... explain what the heck is going on.

The Domain Authentication Process

On the left, find Settings and click "Sender Authentication". We want "Domain Authentication" - click to get started. Ultimately, *all* we will need to do is add a few new records to our domain's DNS. To help make that easier, we can select where we host our DNS settings so that SendGrid can give us instructions customized to that service.

In reality, we haven't deployed our site yet - so we'll walk through this process... for pretend. Let's pretend our DNS is hosted on CloudFlare - I *love* CloudFlare. I'll skip the "link branding" thing - that's something else entirely. Click Next.

Now it wants to know which *domain* we'll send from. Right now, we're sending from `@example.com`. Let's change that to `@thespacebar.com` and pretend that *this* is our production domain. In the box, use `thespacebar.com` and hit "Next".

Here is the important stuff! If you don't care about what's going on, you can simply add these 3 DNS records and skip ahead to where we talk about DMARC. These are enough to *prove* that our SendGrid account is allowed to send emails on behalf of our domain.

But I think this stuff is neat! When it comes to this whole "domain authentication" thing, there are *three* fancy acronyms that you'll hear: SPF, DKIM and DMARC. Here's the 60 second explanation of the first two.

The DNS Settings: SPF & DKIM

Both SFP and DKIM are security mechanisms where you can set specific DNS records that will say exactly *who* is allowed to send emails from your domain. SPF works by whitelisting IP addresses that are allowed to send emails. DKIM works by using a public key to prove that the sender is authorized to send emails. They do similar jobs, but you typically want to have *both*.

Here's what the SPF and DKIM records look like for SymfonyCasts.com:

```
TXT symfonycasts.com                        v=spf1 include:spf.mailjet.com include:
TXT mailjet._domainkey.symfonycasts.com k=rsa; p=MIGfMA0GCSqGSIb3DQEBAQUAA4GNAD
```

The first is the SPF - the sender policy framework. Our framework allows emails to be sent by Mailjet - that's what our site uses for emails - and Helpscout, which is our ticketing system. The second is for DKIM: it lists a public key that can be used to verify that the email *was* really sent by an authorized sender. Your DNS records might looks a bit different, but this is the general idea.

But, wow - the DNS records that SendGrid is telling us to use are *way* different! This is because of a nice "Automated Security" feature they have. The short story is this: by setting these CNAME records, *it* will set up the SPF and DKIM settings for you... which is nice... because they're kinda long, complex strings. If you *do* need more control, on the previous screen we *could* have selected an option to turn "automated security" off. In that case, this step would tell us a couple of `TXT` records we need to set - very similar to the `TXT` records we use for SymfonyCasts.com.

So... DMARC?

The point is: set these DNS records and you're good. But, there is *one* more, *newer* part of email security that is often *not* handled by your cloud email system. It's called DMARC and it's *totally* optional. Here's what the DMARC DNS record looks like for SymfonyCasts:

```
TXT _dmarc v=DMARC1; p=none; pct=100; rua=mailto:re+eymg4cd5p5c@dmarc.postmarka
```

In a nutshell, DMARC adds even a bit *more* confidence to your emails. This crazy string tells email inboxes a few things. For example, it *specifically* says *what* should happen if an email fails SPF or DKIM. Technically, *just* because an email fails DKIM, for example, it *doesn't* mean that the email will *definitely* go to spam: it's just *one* thing that counts against the email's spam score. But, if you want, you could create a DMARC that clarifies this: for example, instructing that all emails that fail SPF or DKIM should be *rejected*.

It also has one other *fascinating* super power, and this is the part I *love*. SPF and DKIM are scary... because what if you set them up wrong? Or you set them up right today, but then you tweak some DNS settings and accidentally break them? Many of your emails might start going to spam without you even realizing it.

DMARC can solve this, and this is how *we* use it. By setting the `rua` key to an email, you can request that all major ISP's send you reports about how many emails they are receiving from your domain and whether or not SPF and DKIM are aligned. Yep, you'll get a report if something is suddenly misconfigured... and you can even see *who* is trying to send fake emails from your domain!

But, instead of getting these low-level messages into your personal inbox, we use a free service from PostMarkApp. The reports are sent to *them*, and we get a neat, weekly update.

Unfortunately, SendGrid doesn't help you set up DMARC. But *fortunately*, by going to https://dmarc.postmarkapp.com/, you can answer a few short questions and get the exact DMARC record you need.

Phew! Enough email, authentication nerdiness! I'll leave you to update your own DNS records and... I'll change the email `from` back to `@example.com`.

And hey! About this `from` address. Every email from our app will probably be *from* the same address. Can we set this globally? Yes! Let's talk about that and events next.

Chapter 26: Events & Overriding "Recipients"

I want to propose two cool ideas.

First, while we're developing, if we decide to use Mailtrap, great: all of our emails will go there. But if we decide that we want to use SendGrid to send *real* emails while developing... it's a little trickier. For example, whenever you register, you would need to use a *real* email address. Otherwise, the email would never make it to your inbox.

So here's idea number 1: what if, in the `dev` environment only, we globally *override* the "to" of every email and send to ourselves. So even if we registered as `space_cadet@example.com` - the email would *actually* be delivered to our real address: `ryan@symfonycasts.com` for me. That would be cool!

My *second* idea is similar: instead of *manually* setting the `from()` on *every* email object... what if we hook into mailer and set this *globally*. That's less duplication and more consistency.

Hooking into Mailer: MessageEvent

The way to accomplish *both* of these is by leveraging an *event*. Whenever an email is sent through Mailer, internally, it dispatches *one* event called `MessageEvent`. Mailer itself comes with a *couple* of classes that can "listen" to this event. The most interesting one is called `EnvelopeListener`.

Built-in Listener: EnvelopeListener

I'll hit Shift+Shift and look for `EnvelopeListener` so we can see inside. Start by looking for `getSubscribedEvents()`. Yep! This is listening on `MessageEvent`. Here's the idea: *if* you used this class, you could instantiate it and pass a custom sender or a custom array of recipients. Then, whenever an email is sent, the `onMessage()` method would be called and it would *override* that stuff on the email.

I love it! Even though this class lives inside Mailer, Symfony doesn't *activate* it by default: it's not currently being used. In Symfony 4.4, some new config options were been added so you can activate & configure it easily:

```
# config/packages/mailer.yaml
# or config/packages/dev/mailer.yaml for only the dev environment
framework:
  mailer:
    envelope:
      sender: 'sender@example.org'
      recipients: ['redirected@example.org']
```

But in Symfony 4.3, if we want to use this class, we need to activate it manually... which is kinda fun anyways.

So here's the plan: to start, in the development environment only, I want *all* emails to *actually* be sent to `ryan@symfonycasts.com`, *regardless* of the `to()` address on the email.

Setting up the Dev Email

To do this, in `.env`, let's create a *brand* new, shiny environment variable: `DEV_MAIL_RECIPIENT` set to, how about, `someone@example.com`.

```
.env
// ... lines 1 - 53
DEV_MAIL_RECIPIENT=someone@example.com
```

That's not a real email, because each developer should need to copy this variable, open their own `.env.local` file, and customize it to whatever *they* want.

Registering EnvelopeListener in dev Only

Next, we need to register `EnvelopeListener` as a service... but *only* in the `dev` environment: I don't want to change the recipients on production. To do that, in the `config/` directory, create a new file called `services_dev.yaml`. Thanks to that `_dev` part, this will only be loaded in the `dev` environment. At the top, start with the same `_defaults` code that we have on top of our main services file: `services:`, then the magic `_defaults:` to set up some *default* options that we want to apply to *every* service registered in this file. The default config we want is `autowire: true` and `autoconfigure: true`.

```yaml
config/services_dev.yaml
services:
    _defaults:
        autowire: true
        autoconfigure: true
// ... lines 5 - 10
```

Now, let's register `EnvelopeListener` as a service. Copy its namespace, paste, add a `\` then go copy the class name and put that here too.

```yaml
config/services_dev.yaml
services:
// ... lines 2 - 5
    Symfony\Component\Mailer\EventListener\EnvelopeListener:
// ... lines 7 - 10
```

For arguments, the class has two: `$sender` and an array of `$recipients`. We'll focus on setting the "sender" globally in a few minutes... but for right now, I *don't* want to use that feature... so we can set the argument to `null`. Under arguments, use `- null` for sender and, for recipients, `- []` with one email inside. To reference the environment variable we created, say `%env()%`, then copy the variable name - `DEV_MAIL_RECIPIENT` - and paste it in the middle.

```yaml
config/services_dev.yaml
services:
// ... lines 2 - 5
    Symfony\Component\Mailer\EventListener\EnvelopeListener:
        arguments:
            - null
            - ['%env(DEV_MAIL_RECIPIENT)%']
```

That should be it! This will register the service and, thanks to `autoconfigure`, Symfony will configure it as an event subscriber.

Testing time! Move over, refresh and... ah! I have a typo! The key should be `_defaults` with an "s". Try it again. This time register with a fake email: `thetruthisoutthere13@example.com`, any password, agree to the terms and register!

Because our app is configured to use SendGrid... that *should* have sent a *real* email. Check the inbox - we have a new one! That's the original email from a minute ago on top... and here's the new one.

Recipients Versus To

But! This is even cooler. If you were watching *really* closely, you may have noticed that, in `EnvelopeListener`, what we're *setting* is something called "recipients". But when we create an email... we use a method call `->to()`. It turns out, those are two different concepts. Gasp!

Back over in gmail, I'll click to view the "original" message. Check this out: this email is *to* `thetruthisoutthere13@example.com`. Search for `ryan@symfonycasts`. Hmm, it says `Delivered-To: ryan@symfonycasts.com`.

Envelope Versus Message

Here's what's going on. *Just* like how, in the real world, you put a "message" into an "envelope" and then send it through the *real-world* mail, an email is *also* these same two parts: the message itself and an *envelope* that goes around it. The `To` of an email is what's written on top of the "message". But the *envelope* around that message could have a totally *different* address on it. *That* is known as the "recipient". The envelope is how the email is *delivered*. And the message is basically what you're looking at inside your inbox.

So by setting the recipients, we changed the address on the envelope, which caused the email to be *delivered* to `ryan@symfonycasts.com`. But the `To` on the message inside is still `thetruthisoutthere13@example.com`.

This... for the most part... is just fun mail trivia. *Most* of the time, the "To" and the "recipients" will be the same. And... that's exactly what happens if you set the `To` but *don't* set the recipients: mailer sets the recipients *for* you... to match the `To`.

This idea becomes even *more* important when we talk about setting the `from` address globally so we don't need to set it on every email. Because... yep, `from` is different than "sender". That's next.

Chapter 27: Setting "From" Globally

I don't like to have this `->from()` on every single email that I create. This will probably *always* be the same, so let's set it globally.

We know that Mailer dispatches an event each time it sends an email. So, we could probably create a *listener* for that event and set the `from` address from there!

But wait. A minute ago, we configured `EnvelopeListener` as a service in the `dev` environment and used it to globally override the recipients. This class *also* allows us to pass a "sender" as the first argument. If we did, it would override the sender on this "envelope" thing.

So, is setting the `from` globally as easy as passing a value to the first argument of `EnvelopeListener`? Is this video about 10 seconds from being over?

From Versus Sender

Sadly... no. Remember when I mentioned that an email is two parts: a message and then an envelope around that message? When you set the `->to()` on an Email, that goes into the message. The *recipients* is what goes on the *envelope*... which *totally* impacts *where* the email is delivered, but does *not* impact who the email *appears* to be addressed to when reading the email.

The same is true when it comes to `from()` versus "sender". But this... is even more subtle. The "sender" is the address that's written on the *envelope* and the `from` is what *actually* goes into the message - this is the part that the user will see when reading the email. It's a weird distinction: it's like if someone mailed a letter on your behalf: *they* would be the sender - with *their* address on the envelope. But when you opened the envelope, the message inside would be signed *from* you.

The *point* is, setting the "sender" is not enough. When we set the `from()`, Mailer *does* automatically use that to set the "sender" on the envelope... unless it was set explicitly. But it does *not* do it the other way around: if we removed the `->from()` line and only set the sender, Mailer would give us a huge error because the message would have *no* from.

So what does this all mean? It means `EnvelopeListener` can't help us: we need to override the "from", not the "sender". No problem: let's create our own event listener.

Creating the Event Subscriber

In the `src/` directory, create a new directory called `EventListener`. And inside, a new PHP class called `SetFromListener`. Make this implement `EventSubscriberInterface`: the interface for all subscribers. I'll go to the "Code -> Generate" menu - or Command + N on a Mac - and hit "Implement Methods" to add the one method required by this interface: `getSubscribedEvents()`.

```
src/EventListener/SetFromListener.php
// ... lines 1 - 2
namespace App\EventListener;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
// ... lines 6 - 9
class SetFromListener implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
// ... lines 14 - 16
    }
// ... lines 18 - 27
}
```

Inside, return an array: we want to listen to `MessageEvent`. So: `MessageEvent::class => 'onMessage'`. When this event occurs, call the `onMessage` method... which we need to create!

```
src/EventListener/SetFromListener.php
// ... lines 1 - 2
namespace App\EventListener;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\Mailer\Event\MessageEvent;
// ... lines 7 - 9
class SetFromListener implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return [
            MessageEvent::class => 'onMessage',
        ];
    }
// ... lines 18 - 27
}
```

On top, add `public function onMessage()`. Because we're listening to `MessageEvent`, *that* will be the first argument: `MessageEvent $event`.

```
src/EventListener/SetFromListener.php
// ... lines 1 - 9
class SetFromListener implements EventSubscriberInterface
{
// ... lines 12 - 18
    public function onMessage(MessageEvent $event)
    {
// ... lines 21 - 26
    }
}
```

So... what's inside of this event object anyways? Surprise! The original Email! Ok, maybe that's not *too* surprising. Add `$email = $event->getMessage()`.

```
src/EventListener/SetFromListener.php
// ... lines 1 - 9
class SetFromListener implements EventSubscriberInterface
{
// ... lines 12 - 18
    public function onMessage(MessageEvent $event)
    {
        $email = $event->getMessage();
// ... lines 22 - 26
    }
}
```

But... is that... *truly* our original Email object... or is it something else? Hold Command or Ctrl and click the `getMessage()` method to jump inside. Hmm, this returns something called a `RawMessage`. What's that?

We have been working with `Email` objects or `TemplatedEmail` objects. Open up `TemplatedEmail` and... let's dig! `TemplatedEmail` extends `Email`... `Email` extends `Message`... and `Message` extends... ah ha! `RawMessage`!

Oooook. *We* typically work with `TemplatedEmail` or `Email`, but on a really, really low level, all Mailer *really* needs is an instance of `RawMessage`. Let's... close a few files. The point is: when we call `$event->getMessage()`, this will return whatever object was actually passed to the `send()` method... which in our case is always going to be a `TemplatedEmail` object. But just to be safe, let's add if `!$email instanceof Email` - make sure you get the one from the Mime component - just return. This shouldn't happen... but could in theory if a third-party bundle sends emails. If you want to be safe, you could also throw an exception here so you *know* if this happens.

```
src/EventListener/SetFromListener.php
// ... lines 1 - 6
use Symfony\Component\Mime\Email;
// ... lines 8 - 9
class SetFromListener implements EventSubscriberInterface
{
// ... lines 12 - 18
    public function onMessage(MessageEvent $event)
    {
        $email = $event->getMessage();
        if (!$email instanceof Email) {
            return;
        }
// ... lines 25 - 26
    }
}
```

Anyways, now that we're sure this is an `Email` object, we can say `$email->from()`... go steal the `from()` inside `Mailer`... and paste here. Re-type the "S" on `NamedAddress` and hit tab to add its `use` statement on top.

```
src/EventListener/SetFromListener.php
// ... lines 1 - 6
use Symfony\Component\Mime\Email;
use Symfony\Component\Mime\NamedAddress;
// ... line 9
class SetFromListener implements EventSubscriberInterface
{
// ... lines 12 - 18
    public function onMessage(MessageEvent $event)
    {
        $email = $event->getMessage();
        if (!$email instanceof Email) {
            return;
        }

        $email->from(new NamedAddress('alienmailcarrier@example.com', 'The Space Bar'));
    }
}
```

Tip

In Symfony 4.4 and higher, use `new Address()` - it works the same way as the old `NamedAddress`.

That's it! We just *globally* set the from! Back in `Mailer`, delete it from `sendWelcomeMessage()`... and also from the weekly report email.

Testing time! Register with *any* email - because we know that all emails are being delivered to `ryan@symfonycasts.com` in the development environment - any password, hit register and... run over to the inbox!

There it is! Welcome to The Space Bar *from* `alienmailer@example.com`.

Next, sending an email requires a network call... so it's a *heavy* operation. We can speed up the user experience by sending emails asynchronously via Messenger.

Chapter 28: Async Emails with Messenger

Sending an email - like after we complete registration - takes a little bit of time because it involves making a network request to SendGrid. Yep, sending emails is *always* going to be a "heavy" operation. And whenever you're doing something heavy... it means your user is waiting for the response. That's... not the end of the world... but it's not ideal.

So... when a user registers, instead of sending the email immediately, could we send it... later and return the response faster? Of course! Thanks to Symfony's Messenger component, which has first-class integration with Mailer.

Installing & Configuring Messenger

First: in our editor, open `.env.local` and, for simplicity. let's change the `MAILER_DSN` back to use Mailtrap. To install Messenger... you can kinda guess the command. In your terminal, run:

```
composer require messenger
```

Messenger is *super* cool and we have an entire tutorial about it. But, it's also simple to get set up and running. Let's see how.

The recipe for Messenger just did a few things: it created a new `messenger.yaml` configuration file and also added a section in `.env`. Let's go find that.

```
.env
// ... lines 1 - 55
###> symfony/messenger ###
# Choose one of the transports below
# MESSENGER_TRANSPORT_DSN=amqp://guest:guest@localhost:5672/%2f/messages
# MESSENGER_TRANSPORT_DSN=doctrine://default
# MESSENGER_TRANSPORT_DSN=redis://localhost:6379/messages
###
```

Here's the 30 second description of how to get Messenger set up. In order to do some work "later" - like sending an email - you need to configure a "queueing" system where details about that work - called "messages" - will be sent. Messenger calls these transports. Because we're already using Doctrine, the easiest "queueing" system is a database table. Uncomment that `MESSENGER_TRANSPORT_DSN` to use it.

Next, open `config/packages/messenger.yaml` - that's the new config file:

config/packages/messenger.yaml

```yaml
framework:
    messenger:
        # Uncomment this (and the failed transport below) to send failed messages to this transport for later handling.
        # failure_transport: failed

        transports:
            # https://symfony.com/doc/current/messenger.html#transport-configuration
            # async: '%env(MESSENGER_TRANSPORT_DSN)%'
            # failed: 'doctrine://default?queue_name=failed'
            # sync: 'sync://'

        routing:
            # Route your messages to the transports
            # 'App\Message\YourMessage': async
```

and uncomment the transport called `async`.

config/packages/messenger.yaml

```yaml
framework:
    messenger:
// ... lines 3 - 5
        transports:
// ... line 7
            async: '%env(MESSENGER_TRANSPORT_DSN)%'
// ... lines 9 - 16
```

Making Emails Async

Great. As *soon* as you install Messenger, when Mailer sends an email, internally, it will automatically start doing that by *dispatching* a message through Messenger. Hit Shift + Shift to open a class called `SendEmailMessage`.

Specifically, Mailer will create *this* object, put our `Email` message inside, and dispatch it through Messenger.

Now, if we *only* installed messenger, the fact that this is being dispatched through the message bus would make... absolutely no difference. The emails would *still* be handled immediately - or *synchronously*.

But *now* we can tell Messenger to "send" instances of `SendEmailMessage` to our `async` transport *instead* of "handling" them - meaning *delivering* the email - right now. We do that via the `routing` section. Go copy the namespace of the `SendEmailMessage` class and, under `routing`, I'll clear out the comments and say `Symfony\Component\Mailer\Messenger\`, copy the class name, and paste: `SendEmailMessage`. Set this to `async`.

```
config/packages/messenger.yaml
framework:
    messenger:
// ... lines 3 - 11
        routing:
            # Route your messages to the transports
            # 'App\Message\YourMessage': async
            'Symfony\Component\Mailer\Messenger\SendEmailMessage':  async
```

Hey! We just made *all* emails async! Woo! Let's try it: find the registration page.... register as "Fox", email `thetruthisoutthere15@example.com`, any password, agree to the terms and register!

You may not have noticed, but if you compared the response times of submitting the form before and after that change... this was way, *way* faster.

Checking out the Queue

Over in Mailtrap... there are no new messages. I can refresh and... nothing. The email was *not* delivered. Yay! Where is it? Sitting & waiting inside our queue... which is a database table. You can see it by running:

```
php bin/console doctrine:query:sql 'SELECT * FROM messenger_messages'
```

That table was automatically created when we sent our first message. It has one row with our *one* Email inside. If you look closely... you can see the details: the subject, and the email *template* that will be rendered when it's delivered.

Running the Worker

How do we *actually* send the email? In Messenger, you process any waiting messages in the queue by running:

```
php bin/console messenger:consume -vv
```

The `-vv` adds extra debugging info... it's more fun. This process is called a "worker" - and you'll have at least one of these commands running at all times on production. Check out our Messenger tutorial for details about that.

Cool! The message was "received" from the queue and "handled"... which is a fancy way in *this* case to say that the email *was* actually delivered! Go check out Mailtrap! Ah! There it is! The full correct email... in all its glory.

By the way, in order for your emails to be rendered correctly when being sent via Messenger, you need to make sure that you have the route context parameters set up correctly. That's a topic we covered earlier in this tutorial.

So... congrats on your new shiny async emails! Next, let's make sure that the "author weekly report" email still works... because... honestly... there's going to be a gotcha. Also, how does sending to a transport affect our functional tests?

Chapter 29: Attachments with Async Messenger Emails

Our registration email is being sent asynchronously via Messenger. And actually, *every* email our app sends will now be async. Let's double-check that the weekly report emails are still working.

Hit Ctrl+C to stop the worker process and, just to make sure our database if full of *fresh* data, reload the fixtures:

```
php bin/console doctrine:fixtures:load
```

Now run:

```
php bin/console app:author-weekly-report:send
```

Problems with Binary Attachments

Ah! Explosion! Incorrect string value? Wow. Okay. What we're seeing is a real-world limitation of the doctrine transport: it can't handle binary data. This *may* change in Symfony 4.4 - there's a pull request for it - but it may not be merged in time.

Why does our email contain binary data? Remember: the method that creates the author weekly report email *also* generates a PDF and attaches it. That PDF is binary... so when Messenger tries to put it into a column that doesn't support binary data... boom! Weird explosion.

If this is a problem for you, there are two fixes. First, instead of Doctrine, use another transport - like AMQP. Second, if you need to use doctrine and you *do* send binary attachments, instead of saying `->attach()` you can say `->attachFromPath()` and pass this a *path* on the filesystem to the file. By doing this, the *path* to the file is what is stored in the queue. The only caveat is that the worker needs to have access to the file at that path.

Messenger and Tests

There's one other thing I want to show with messenger. Run the tests!

```
php bin/phpunit
```

Awesome! There are a *bunch* of deprecation notices, but the tests *do* pass. However, run that Doctrine query again to see the queue:

```
php bin/console doctrine:query:sql 'SELECT * FROM messenger_messages'
```

Uh oh... the email - the one from our functional test to the registration page - was added to the queue! Why is that a problem? Well, it's not a *huge* problem... but if we run the `messenger:consume` command...

```
php bin/console messenger:consume -vv
```

That would actually send that email! Again, that's not the end of the world... it's just a little odd - the test environment doesn't need to send real emails.

If you've configured your `test` environment to use a different database than normal, you're good: your test database queue table *will* fill up with messages, but you'll never run the `messenger:consume` command from that environment anyways.

Overriding the Transport in the test Environment

But there's also a way to solve this directly in Messenger. In `.env`, copy `MESSENGER_TRANSPORT_DSN` and open up `.env.test`. Paste this but replace `doctrine` with `in-memory`. So: `in-memory://`

```
.env.test
// ... lines 1 - 4
MESSENGER_TRANSPORT_DSN=in-memory://default
```

This transport... is useless! And I *love* it. When Messenger sends something to an "in-memory" transport, the message... actually goes nowhere - it's just discarded.

Run the tests again:

```
php bin/phpunit
```

And... check the database:

```
php bin/console doctrine:query:sql 'SELECT * FROM messenger_messages'
```

No messages! Next, lets finish our grand journey through Mailer by integrating our Email styling with Webpack Encore.

Chapter 30: Styling Emails with Encore & Sass Part 1

Our app uses Webpack Encore to manage its frontend assets. It's not something we talked much about because, if you downloaded the course code from this page, it already included the final `build/` directory. I did this so we didn't need to worry about setting up Encore *just* to get the site working.

But if you *are* using Encore, we can make a few improvements to how we're styling our emails. Specifically, we took *two* shortcuts. First, the `assets/css/foundation-emails.css` file is something we downloaded from the Foundation website. That's not how we would *normally* do things with Encore. If we need to use a third-party library, we typically install it with `yarn` instead of committing it directly.

The other shortcut was with this `emails.css` file. I'd *rather* use Sass... but to do that, I need to process it through Encore.

Installing Foundation Emails via Yarn

Let's get to work! Over in the terminal, start by installing all the current Encore dependencies with:

```
yarn install
```

When that finishes, install Foundation for Emails with:

```
yarn add foundation-emails --dev
```

The *end* result is that we now have a giant `node_modules/` directory and... somewhere *way* down in this giant directory... we'll find a `foundation-emails` directory with a `foundation-emails.css` file inside. They also have a Sass file if you want to import *that* and control things further... but the CSS file is good enough for us.

Before we make any real changes, make sure Encore can build by running:

```
yarn dev --watch
```

And... excellent! Everything is working.

Using Sass & Importing Foundation Emails

Now that we've installed Foundation for Emails properly, let's delete the committed file: I'll right click and go to "Refactor -> Delete". Next, because I want to use Sass for our custom email styling, right click on `email.css`, go to "Refactor -> Rename" and call it `email.scss`.

Because this file will be processed through Encore, we can import the `foundation-email.css` file from right here with `@import`, a `~` - that tells Webpack to look in the `node_modules/` directory - then `foundation-emails/dist/foundation-emails.css`.

assets/css/email.scss

```
@import "~foundation-emails/dist/foundation-emails.css";

// ... lines 3 - 39
```

This feels good! I'll close up `node_modules/`... cause it's giant.

Creating the Email Entry

Now open up the email layout file: `templates/email/emailBase.html.twig`. When we used `inline_css()`, we pointed it at the `foundation-emails.css` file *and* the `email.css` file. But now... we only really need to point it at `email.scss`... because, in theory, that will include the styles from *both* files.

```
templates/email/emailBase.html.twig
{% apply inky_to_html|inline_css(source('@styles/foundation-emails.css'),
source('@styles/email.css')) %}
// ... lines 2 - 30
{% endapply %}
```

The problem is that this is now a *Sass* file... and `inline_css` only works with CSS files: we can't point it at a Sass file and expect it transform the Sass into CSS. And even if it *were* a CSS file, the `@import` won't work unless we process this through Encore.

So here's the plan: we're going to pretend that `email.scss` is just an ordinary CSS file that we want to include on some page on our site. Open up `webpack.config.js`. Whenever we have some page-specific CSS or JS, we add a new *entry* for it. In this case, because we don't need any JavaScript, we can add a "style" entry. Say `.addStyleEntry()` - call the entry, how about, `email`, and point it at the file: `./assets/css/email.scss`.

```
webpack.config.js
// ... lines 1 - 2
Encore
// ... lines 4 - 24
    .addStyleEntry('email', './assets/css/email.scss')
    //.addEntry('page1', './assets/js/page1.js')
// ... lines 27 - 77
;
// ... lines 79 - 80
```

To get Webpack to see the updated config, in the terminal, press Ctrl+C to stop Encore and restart it:

```
yarn dev --watch
```

And... it builds! Interesting: the `email` entrypoint dumped *two* CSS files. Let's look at the `public/build` directory. Yep: `email.css` and also this `vendors~email.css`.

This is thanks to an optimization that Wepback Encore makes when you use `splitEntryChunks()`... which you can learn *all* about in our Encore tutorial. But the basic point is that if we want *all* of the CSS from the built `email.scss` file, we need to include *both* `email.css` *and* `vendor~email.css`.

Ok, easy, right? In the template, we could load the source of `vendor~email.css` and `email.css`. The *problem* is that Webpack splits the files in a very dynamic fashion: if it finds a more efficient way to split the files tomorrow - maybe into *three* files - it will! Plus, when we do our production build, the files will include a dynamic *hash* in their filename - like `email.123abc.css`.

So... we need to do a bit more work to reliably load this stuff through `inline_css()`. Let's do that next with a custom Twig function.

Chapter 31: Processing Encore Files through inline_css()

We just used Encore to build an `email.scss` file that we want to process through `inline_css()` to style our emails. The *problem* is that, instead of building just *one* `email.css` file in `public/build`, it split it into two for performance reasons. That wouldn't be a problem, except that the *way* Webpack splits the files might change over time - we can't guarantee that it will *always* be these two files. To make matters worse, an Encore production build will add a dynamic "hash" to every file - like `email.123abc.css`.

Basically... pointing `inline_css()` directly at these two files... isn't going to work.

How Dynamic Files are Normally Rendered

This is why, in `base.html.twig` we simply use `encore_entry_link_tags()` and it takes care of everything. How? Behind the scenes, it looks in the `public/build/` directory for an `entrypoints.json` file that Encore builds. This is the *key*: it tells us *exactly* which CSS and JS files are needed for each entrypoint - like `app`. Or, for `email`, yep! It contains the two CSS files.

The *problem* is that we don't want to just output `link` tags. We actually need to read the *source* of those files and pass *that* to `inline_css()`.

Let's create a new Twig Function!

Since there's no built-in way to do that, let's make our *own* Twig function where we can say `encore_entry_css_source()`, pass it `email`, and *it* will figure out all the CSS files it needs, load their contents, and return it as one big, giant, beautiful string.

templates/email/emailBase.html.twig

```
{% apply inky_to_html|inline_css(encore_entry_css_source('email')) %}
// ... lines 2 - 30
{% endapply %}
```

To create the function, our app already has a Twig extension called `AppExtension`. Inside, say `new TwigFunction()`, call it `encore_entry_css_source` and when this function is used, Twig should call a `getEncoreEntryCssSource` method.

```
src/Twig/AppExtension.php
// ... lines 1 - 13
class AppExtension extends AbstractExtension implements
ServiceSubscriberInterface
{
// ... lines 16 - 24
    public function getFunctions(): array
    {
        return [
// ... line 28
            new TwigFunction('encore_entry_css_source', [$this,
'getEncoreEntryCssSource']),
        ];
    }
// ... lines 32 - 75
}
```

Copy that name and create it below: `public function getEncoreEntryCssSource()` with a `string $entryName` argument. This will return the `string` CSS source.

```
src/Twig/AppExtension.php
// ... lines 1 - 13
class AppExtension extends AbstractExtension implements
ServiceSubscriberInterface
{
// ... lines 16 - 53
    public function getEncoreEntryCssSource(string $entryName): string
    {
// ... lines 56 - 65
    }
// ... lines 67 - 75
}
```

Inside, we need to look into the `entrypoints.json` file to find the CSS filenames needed for this `$entryName`. Fortunately, Symfony has a service that already does that. We can get it by using the `EntrypointLookupInterface` type-hint.

For reasons I don't want to get into in this tutorial, instead of using normal constructor injection - where we add an argument type-hinted with `EntrypointLookupInterface` - we're using a "service subscriber". You can learn about this in, oddly-enough, our tutorial about Symfony & Doctrine.

To fetch the service, go down to `getSubscribedServices()` and add `EntrypointLookupInterface::class`.

src/Twig/AppExtension.php

```php
// ... lines 1 - 8
use Symfony\WebpackEncoreBundle\Asset\EntrypointLookupInterface;
// ... lines 10 - 13
class AppExtension extends AbstractExtension implements ServiceSubscriberInterface
{
// ... lines 16 - 67
    public static function getSubscribedServices()
    {
        return [
// ... lines 71 - 72
            EntrypointLookupInterface::class,
        ];
    }
}
```

Back up in `getEncoreEntryCssSource()`, we can say `$files = $this->container->get(EntrypointLookupInterface::class)` - that's how you access the service using a service subscriber - then `->getCssFiles($entryName)`.

src/Twig/AppExtension.php

```php
// ... lines 1 - 13
class AppExtension extends AbstractExtension implements ServiceSubscriberInterface
{
// ... lines 16 - 53
    public function getEncoreEntryCssSource(string $entryName): string
    {
        $files = $this->container
            ->get(EntrypointLookupInterface::class)
            ->getCssFiles($entryName);
// ... lines 59 - 65
    }
// ... lines 67 - 75
}
```

This will return an array with something like these two paths. Next, `foreach` over `$files as $file` and, above create a new `$source` variable set to an empty string. All we need to do now is look for each file inside the `public/` directory and fetch its contents.

```php
// src/Twig/AppExtension.php
// ... lines 1 - 13
class AppExtension extends AbstractExtension implements ServiceSubscriberInterface
{
// ... lines 16 - 53
    public function getEncoreEntryCssSource(string $entryName): string
    {
        $files = $this->container
            ->get(EntrypointLookupInterface::class)
            ->getCssFiles($entryName);

        $source = '';
        foreach ($files as $file) {
// ... line 62
        }
// ... lines 64 - 65
    }
// ... lines 67 - 75
}
```

Adding a publicDir Binding

We *could* hardcode the path to the `public/` directory right here. But instead, let's set up a new "binding" that we can pass through the constructor. Open up `config/services.yaml`. In our Symfony Fundamentals Course, we talk about how the global `bind` below `_defaults` can be used to allow scalar arguments to be autowired into our services. Add a new one: `string $publicDir` set to `%kernel.project_dir%` - that's a built-in parameter - `/public`.

```
config/services.yaml
// ... lines 1 - 12
services:
// ... line 14
    _defaults:
// ... lines 16 - 22
        bind:
// ... lines 24 - 27
            string $publicDir: '%kernel.project_dir%/public'
// ... lines 29 - 54
```

This `string` part before `$publicDir` is optional. But by adding it, we're *literally* saying that this value should be passed if an argument is exactly `string $publicDir`. Being able to add the type-hint to a bind is a new feature in Symfony 4.2. We didn't use it on the earlier binds... but we could have.

Back in `AppExtension`, add the `string $publicDir` argument. I'll hit "Alt + Enter" and go to "Initialize fields" to create that property and set it.

```
src/Twig/AppExtension.php
// ... lines 1 - 13
class AppExtension extends AbstractExtension implements ServiceSubscriberInterface
{
// ... line 16
    private $publicDir;
// ... line 18
    public function __construct(ContainerInterface $container, string $publicDir)
    {
// ... line 21
        $this->publicDir = $publicDir;
    }
// ... lines 24 - 75
}
```

Down in the method, we can say `$source .= file_get_contents($this->publicDir.$file)` - each `$file` path should already have a `/` at the beginning. Finish the method with `return $source`.

Tip

To avoid missing CSS if you send your emails via Messenger (or if you send multiple emails during the same request), "reset" Encore's internal cache before calling `getCssFiles()`:

```
// replace the first 3 lines with these
$entryPointLookupInterface = $this->container->get(EntrypointLookupInterface
$entryPointLookupInterface->reset();
$files = $entryPointLookupInterface->getCssFiles($entryName);

$source = '';
// ...
```

src/Twig/AppExtension.php

```
// ... lines 1 - 13
class AppExtension extends AbstractExtension implements
ServiceSubscriberInterface
{
// ... lines 16 - 53
    public function getEncoreEntryCssSource(string $entryName): string
    {
        $files = $this->container
            ->get(EntrypointLookupInterface::class)
            ->getCssFiles($entryName);

        $source = '';
        foreach ($files as $file) {
            $source .= file_get_contents($this->publicDir.'/'.$file);
        }

        return $source;
    }
// ... lines 67 - 75
}
```

Whew! Let's try this! We're already running Encore... so it already dumped the `email.css` and `vendors~email.css` files. Ok, let's go send an email. I'll hit back to get to the registration page, bump the email, type any password, hit register and... wow! No errors! Over in Mailtrap... nothing here... Of course! We refactored to use Messenger... so emails are *not* sent immediately!

By the way, if that *annoys* you in development, there *is* a way to handle async messages immediately while coding. Check out the Messenger tutorial.

Let's start the worker and send the email. I'll open another tab in my terminal and run:

```
php bin/console messenger:consume -vv
```

Message received... and... message handled. Go check it out! The styling look great: they're inlined and coming from a proper Sass file.

And... we've made it to the end! You are now an email *expert*... I mean, not just a Mailer expert... we *really* dove deep. Congrats!

Go forth and use your great power responsibly. Let us know what cool emails you're sending... heck... you could even *send* them to us... and, as always, we're here to help down in the comments section.

Alright friends, seeya next time!

With <3 from SymfonyCasts