Mailer y Webhook con Mailtrap

Chapter 1: Instalar el Mailer

¡Hola amigos! ¡Bienvenidos a "Symfony Mailer con Mailtrap"! Soy Kevin, y seré tu postmaster para este curso, que trata sobre el envío de correos electrónicos bonitos con el componente Mailer de Symfony, incluyendo la adición de HTML, CSS - y la configuración para producción. En ese sentido, hay muchos servicios que puedes utilizar en producción para enviar tus correos electrónicos. Este curso se centrará en uno llamado Mailtrap: (1) porque es genial y (2) porque ofrece una forma fantástica de previsualizar tus correos electrónicos. Pero no te preocupes, los conceptos que trataremos son universales y pueden aplicarse a cualquier servicio de correo electrónico. ¡Y además! También veremos cómo rastrear eventos de correo electrónico como rebotes, aperturas y clics en enlaces aprovechando algunos componentes relativamente nuevos de Symfony: Webhook y RemoteEvent.

Correos electrónicos transaccionales vs masivos

Antes de empezar a enviar información importante por correo electrónico, tenemos que aclarar algo: Symfony Mailer es sólo para lo que se llama correos electrónicos transaccionales. Son correos específicos de usuario que se producen cuando ocurre algo concreto en tu aplicación. Cosas como: un correo electrónico de bienvenida después de que un usuario se registre, un correo electrónico de confirmación de pedido cuando realizan un pedido, o incluso correos electrónicos como "tu post ha sido votado" son ejemplos de correos electrónicos transaccionales. Symfony Mailer no es para emails masivos o de marketing. Por ello, no tenemos que preocuparnos de ningún tipo de funcionalidad para darse de baja. Existen servicios específicos para enviar correos masivos o boletines informativos, Mailtrap incluso puede hacerlo a través de su sitio web.

Nuestro proyecto

Como siempre, para sacar el máximo partido a tu dinero en screencast, ¡deberías codificar conmigo! Descarga el código del curso en esta página. Cuando descomprimas el archivo, encontrarás un directorio `start/` con el código con el que empezaremos. Sigue el archivo `README.md` para poner en marcha la aplicación. Yo ya lo he hecho y he ejecutado `symfony serve -d` para iniciar el servidor web.

Bienvenido a "Viajes Universales": una agencia de viajes donde los usuarios pueden reservar viajes a diferentes lugares galácticos. Aquí tienes los viajes disponibles actualmente. Los usuarios ya pueden reservarlos, pero no se envían correos electrónicos de confirmación cuando lo hacen. ¡Vamos a arreglar eso! Si voy a gastar miles de créditos en un viaje a Naboo, ¡quiero saber que mi reserva se ha realizado correctamente!

Instalar el componente Mailer

Paso 1: ¡instalemos el Mailer de Symfony! Abre tu terminal y ejecuta:

```
composer require mailer
```

La receta de Symfony Flex para el mailer nos pide que instalemos alguna configuración de Docker. Esto es para un servidor SMTP local que nos ayude con la previsualización de los correos electrónicos. Vamos a utilizar Mailtrap para esto, así que di "no". ¡Instalado! Ejecuta::

```
git status
```

para ver lo que tenemos. Parece que la receta añadió algunas variables de entorno en `.env` y añadió la configuración del mailer en `config/packages/mailer.yaml`.

`MAILER_DSN`

En tu IDE, abre `.env`. La receta del Mailer añadió esta variable de entorno `MAILER_DSN`. Se trata de una cadena especial con aspecto de URL que configura el transporte de tu mailer: cómo se envían realmente tus correos electrónicos, por ejemplo a través de SMTP, Mailtrap, etc. La

receta utiliza por defecto `null://null` y es perfecta para el desarrollo local y las pruebas. Este transporte no hace nada cuando se envía un correo electrónico Finge entregar el correo electrónico, pero en realidad lo envía por una esclusa de aire. Previsualizaremos nuestros correos electrónicos de otra forma.

¡Vale! ¡Estamos listos para enviar nuestro primer correo electrónico! ¡Hagámoslo a continuación!

Chapter 2: Enviar nuestro primer correo electrónico

¡Vamos de viaje! "Visitar Krypton", ¡Esperemos que aún no haya sido destruido! Sin molestarme en comprobarlo, ¡reservémoslo! Utilizaré el nombre: "Kevin", el correo electrónico "kevin@example.com" y una fecha cualquiera en el futuro. Pulsa "Reservar viaje".

Esta es la página de "detalles de la reserva". Fíjate en la URL: tiene un token único específico para esta reserva. Si un usuario necesita volver aquí más tarde, actualmente, tiene que marcar esta página o enviarse a sí mismo la URL si es como yo ¡Lamentable! Enviémosles un correo electrónico de confirmación que incluya un enlace a esta página.

Quiero que esto ocurra después de guardar la reserva por primera vez. Abre `TripController` y busca el método `show()`. Esto hace la reserva: si el formulario es válido, crea o recupera un cliente y crea una reserva para este cliente y viaje. Luego redirigimos a la página de detalles de la reserva. Deliciosamente aburrido hasta ahora, tal y como me gusta mi código, y los fines de semana.

Inyecta `MailerInterface`

Quiero enviar un correo electrónico después de crear la reserva. Date un poco de espacio moviendo cada argumento del método a su propia línea. Después, añade `MailerInterface $mailer` para obtener el servicio principal de envío de correos electrónicos:

src/Controller/TripController.php

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

Crear el correo electrónico

Después de `flush()`, que inserta la reserva en la base de datos, crea un nuevo objeto de correo electrónico: `$email = new Email()` (el de `Symfony\Component\Mime`). Envuélvelo entre paréntesis para que podamos encadenar métodos. ¿Qué necesita cada correo electrónico? Una dirección de correo electrónico `from`: `->from()` qué tal `info@univeral-travel.com`. Una dirección de correo electrónico `to`: `->to($customer->getEmail())`. Ahora, el `subject`: `->subject('Booking Confirmation')`. Y por último, el correo electrónico necesita un cuerpo: `->text('Your booking has been confirmed')` - suficiente por ahora:

src/Controller/TripController.php

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

Envía el correo electrónico

Termina con `$mailer->send($email)`:

src/Controller/TripController.php

```
// ... lines 1 - 18
final class TripController extends AbstractController
{
// ... lines 21 - 29
    public function show(
// ... lines 31 - 35
    ): Response {
// ... lines 37 - 38
        if ($form->isSubmitted() && $form->isValid()) {
// ... lines 40 - 55
            $mailer->send($email);
// ... lines 57 - 58
        }
// ... lines 60 - 64
    }
}
```

¡Vamos a probarlo!

De nuevo en nuestra aplicación, vuelve a la página de inicio y elige un viaje. Para el nombre, utiliza "Steve", correo electrónico, "steve@minecraft.com", cualquier fecha en el futuro, y reserva el viaje.

Vale... esta página tiene exactamente el mismo aspecto que antes. ¿Se ha enviado un correo electrónico? Nada en la barra de herramientas de depuración web parece indicarlo...

En realidad, el correo electrónico se envió en la petición anterior: el envío del formulario. Ese controlador nos redirigió a esta página. Pero la barra de herramientas de depuración web nos ofrece un atajo para acceder al perfilador de la petición anterior: pasa el ratón por encima de `200` y haz clic en el enlace del perfilador para acceder a él.

Correo electrónico en el perfilador

Echa un vistazo a la barra lateral: ¡tenemos una nueva pestaña "Correos electrónicos"! Y muestra que se ha enviado 1 correo electrónico. ¡Lo hicimos! ¡Haz clic en él y aquí está nuestro correo electrónico! Los campos "De", "Para", "Asunto" y "Cuerpo" son los esperados.

Recuerda que estamos utilizando el transporte de correo `null`, así que este correo no se ha enviado realmente, ¡pero es genial que podamos previsualizarlo en el perfilador!

Aunque... Creo que ambos sabemos que este correo... es... bastante cutre. ¡No da ninguna información útil! ¡Ni URL a la página de detalles de la reserva, ni destino, ni fecha, ni nada! Es tan inútil que me alegro de que el transporte `null` lo tire por la ventana espacial.

¡Eso a continuación!

Chapter 3: Un correo electrónico mejor

Creo que tú, yo, cualquiera que haya recibido alguna vez un correo electrónico, podemos estar de acuerdo en que nuestro primer correo electrónico apesta. No aporta ningún valor. ¡Mejorémoslo!

`Address` Objeto

En primer lugar, podemos añadir un nombre al correo electrónico. Esto aparecerá en la mayoría de los clientes de correo electrónico en lugar de sólo la dirección de correo electrónico: tiene un aspecto más fluido. Envuelve el `from` con `new Address()`, el de `Symfony\Component\Mime`. El primer argumento es el correo electrónico, y el segundo es el nombre: ¿qué tal `Universal Travel`:

src/Controller/TripController.php

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

También podemos envolver el `to` con `new Address()`. y pasar `$customer->getName()` para el nombre:

src/Controller/TripController.php

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

Para el `subject`, añade el nombre del viaje: `'Booking Confirmation for ' . $trip->getName()`:

src/Controller/TripController.php

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

Para el cuerpo `text`. Podríamos alinear todo el texto aquí. Eso se pondría feo, así que ¡utilicemos Twig! Necesitamos una plantilla. En `templates/`, añade un nuevo directorio `email/` y, dentro, crea un nuevo archivo: `booking_confirmation.txt.twig`. Twig puede utilizarse para cualquier formato de texto, no sólo para `html`. Una buena práctica es incluir el formato - `.html` o `.txt` - en el nombre del archivo. Pero a Twig no le importa eso: es sólo para satisfacer nuestro cerebro humano. Volveremos a este archivo en un segundo.

Plantilla de correo Twig

Vuelve a `TripController::show()`, en lugar de `new Email()`, utiliza `new TemplatedEmail()` (el de `Symfony\Bridge\Twig`):

src/Controller/TripController.php

```
// ... lines 1 - 19
final class TripController extends AbstractController
{
// ... lines 22 - 30
    public function show(
// ... lines 32 - 36
    ): Response {
// ... lines 38 - 39
        if ($form->isSubmitted() && $form->isValid()) {
// ... lines 41 - 49
            $email = (new TemplatedEmail())
// ... lines 51 - 64
        }
// ... lines 66 - 70
    }
}
```

Sustituye `->text()` por `->textTemplate('email/booking_confirmation.txt.twig')`:

src/Controller/TripController.php

```
// ... lines 1 - 19
final class TripController extends AbstractController
{
// ... lines 22 - 30
    public function show(
// ... lines 32 - 36
    ): Response {
// ... lines 38 - 39
        if ($form->isSubmitted() && $form->isValid()) {
// ... lines 41 - 49
            $email = (new TemplatedEmail())
// ... lines 51 - 53
                ->textTemplate('email/booking_confirmation.txt.twig')
// ... lines 55 - 59
            ;
// ... lines 61 - 64
        }
// ... lines 66 - 70
    }
}
```

Para pasar variables a la plantilla, utiliza `->context()` con `'customer' => $customer, 'trip' => $trip, 'booking' => $booking`:

src/Controller/TripController.php

```
// ... lines 1 - 19
final class TripController extends AbstractController
{
// ... lines 22 - 30
    public function show(
// ... lines 32 - 36
    ): Response {
// ... lines 38 - 39
        if ($form->isSubmitted() && $form->isValid()) {
// ... lines 41 - 49
            $email = (new TemplatedEmail())
// ... lines 51 - 54
                ->context([
                    'customer' => $customer,
                    'trip' => $trip,
                    'booking' => $booking,
                ])
            ;
// ... lines 61 - 64
        }
// ... lines 66 - 70
    }
}
```

Ten en cuenta que aquí técnicamente no estamos renderizando la plantilla Twig: Mailer lo hará por nosotros antes de enviar el correo electrónico.

Esto es código Twig normal y aburrido. Vamos a mostrar el nombre del usuario utilizando un truco barato, el nombre del viaje, la fecha de salida y un enlace para gestionar la reserva. Necesitamos utilizar URLs absolutas en los correos electrónicos -como https://univeral-travel.com/booking-, así que aprovecharemos la función Twig `url()` en lugar de `path()`: `{{ url('booking_show', {'uid': booking.uid}) }}`. Terminaremos educadamente con, `Regards, the Universal Travel team`:

templates/email/booking_confirmation.txt.twig

```
Hey {{ customer.name|split(' ')|first }},

Get ready for your trip to {{ trip.name }}!

Departure: {{ booking.date|date('Y-m-d') }}

Manage your booking: {{ url('booking_show', {uid: booking.uid}) }}

Regards,
The Universal Travel Team
```

¡Cuerpo del correo electrónico listo! Pruébalo. De vuelta en tu navegador, elige un viaje, nombre: `Steve`, correo electrónico: `steve@minecraft.com`, cualquier fecha en el futuro, y reserva el viaje. Abre el perfil de la última petición y haz clic en la pestaña `Emails` para ver el correo electrónico.

¡Mucho mejor! Observa que las direcciones `From` y `To` ahora tienen nombre. ¡Y nuestro contenido de texto es definitivamente más valioso! Copia la URL de la reserva y pégala en tu navegador para asegurarte de que va al lugar correcto. Parece que sí, ¡bien!

A continuación, utilizaremos la herramienta de pruebas de Mailtrap para obtener una vista previa más robusta del correo electrónico.

Chapter 4: Previsualizar correos electrónicos con Mailtrap (Pruebas de correo electrónico)

Previsualizar correos electrónicos en el perfilador está bien para correos básicos, pero pronto añadiremos estilos HTML e imágenes de gatos espaciales. Para ver correctamente el aspecto de nuestros correos electrónicos, necesitamos una herramienta más robusta. Vamos a utilizar la herramienta de prueba de correo electrónico de Mailtrap. Esto nos proporciona un servidor SMTP real al que podemos conectarnos, pero en lugar de entregar los correos electrónicos a bandejas de entrada reales, ¡van a una bandeja de entrada falsa que podemos comprobar! Es como si enviáramos un correo electrónico de verdad y luego pirateáramos la cuenta de esa persona para verlo... ¡pero sin las molestias ni todas esas cosas ilegales!

Bandeja de entrada falsa

Ve a https://mailtrap.io y regístrate para obtener una cuenta gratuita. Su plan gratuito tiene algunos límites, pero es perfecto para empezar. Una vez dentro, estarás en la página de inicio de su aplicación. Lo que nos interesa ahora es probar el correo electrónico, así que haz clic en él. Deberías ver algo así. Si aún no tienes una bandeja de entrada, añade una aquí.

Abre esa nueva y brillante bandeja de entrada. A continuación, tenemos que configurar nuestra aplicación para que envíe correos electrónicos a través del servidor SMTP Mailtrap. Esto es muy fácil Aquí abajo, en "Ejemplos de código", haz clic en "PHP" y luego en "Symfony". Copia el archivo `MAILER_DSN`.

`MAILER_DSN` para Bandeja de entrada falsa

Como se trata de un valor sensible, y puede variar entre desarrolladores, no lo añadas a `.env`, ya que está compilado en git. En su lugar, crea un nuevo archivo `.env.local`en la raíz de tu proyecto. Pega aquí `MAILER_DSN` para anular el valor de `.env`.

¡Ya estamos preparados para probar Mailtrap! ¡Ha sido fácil! ¡A probar!

De vuelta en la aplicación, reserva un nuevo viaje: Nombre: `Steve`, Email: `steve@minecraft.com`, cualquier fecha en el futuro, y... ¡reserva! Esta petición tarda un poco más porque se está conectando al servidor SMTP externo Mailtrap.

Correo electrónico en Mailtrap

De vuelta en Mailtrap, ¡bam! ¡El correo electrónico ya está en nuestra bandeja de entrada! Haz clic para comprobarlo. Aquí tienes una vista previa "Texto" y una vista "Sin procesar". También hay un "Análisis de Spam" - ¡genial! la "Información técnica" muestra todas las "cabeceras de correo electrónico" en un formato fácil de leer.

Estas pestañas "HTML" están en gris porque no tenemos una versión HTML de nuestro correo electrónico... todavía... ¡Cambiemos eso a continuación!

Chapter 5: Correos electrónicos en HTML

Los correos electrónicos siempre deben tener una versión en texto plano, pero también pueden tener una versión en HTML. ¡Y ahí es donde está la diversión! ¡Es hora de hacer este correo electrónico más presentable añadiéndole HTML!

Plantilla de correo electrónico HTML

En `templates/email/`, copia `booking_confirmation.txt.twig` y nómbrala `booking_confirmation.html.twig`. La versión HTML actúa un poco como una página HTML completa. Envuélvelo todo en una etiqueta `<html>`, añade una `<head>` vacía y envuelve el contenido en una `<body>`. También envolveré estas líneas en etiquetas `<p>` para conseguir algo de espaciado... y una etiqueta `<br>` después de "Saludos", para añadir un salto de línea.

Ahora esta URL puede vivir en una etiqueta `<a>` adecuada. Déjate algo de espacio y corta "Gestiona tu reserva". Añade una etiqueta`<a>` con la URL como atributo `href` y pega el texto dentro.

templates/email/booking_confirmation.html.twig

```
<html>
<head></head>
<body>
<p>Hey {{ customer.name|split(' ')|first }},</p>

<p>Get ready for your trip to {{ trip.name }}!</p>

<p>Departure: {{ booking.date|date('Y-m-d') }}</p>

<p>
    <a href="{{ url('booking_show', {uid: booking.uid}) }}">
        Manage your booking
    </a>
</p>

<p>
    Regards,<br>
    The Universal Travel Team
</p>
</body>
</html>
```

Por último, tenemos que decirle a Mailer que utilice esta plantilla HTML. En `TripController::show()`, encima de `->textTemplate()`, añade `->htmlTemplate()` con `email/booking_confirmation.html.twig`:

src/Controller/TripController.php

```
// ... lines 1 - 19
final class TripController extends AbstractController
{
// ... lines 22 - 30
    public function show(
// ... lines 32 - 36
    ): Response {
// ... lines 38 - 49
            $email = (new TemplatedEmail())
// ... lines 51 - 53
                ->htmlTemplate('email/booking_confirmation.html.twig')
                ->textTemplate('email/booking_confirmation.txt.twig')
// ... lines 56 - 60
            ;
// ... lines 62 - 65
        }
// ... lines 67 - 71
    }
}
```

Pruébalo reservando un viaje: `Steve`, `steve@minecraft.com`, cualquier fecha en el futuro, reserva... y luego comprueba Mailtrap. El correo electrónico tiene el mismo aspecto, ¡pero ahora tenemos una pestaña HTML!

Ah, y la "Comprobación de HTML" está muy bien. Te da un indicador de qué porcentaje de clientes de correo electrónico admiten el HTML de este correo. Por si no lo sabías, los clientes de correo electrónico son un coñazo: es como volver a los 90 con distintos navegadores. Esta herramienta te ayuda con eso.

De nuevo en la pestaña HTML, haz clic en el enlace para asegurarte de que funciona. ¡Funciona!

Así que ahora nuestro correo electrónico tiene una versión en texto y otra en HTML, pero... es un poco pesado mantener ambas. De todas formas, ¿quién utiliza un cliente de correo electrónico sólo de texto? Probablemente nadie o un porcentaje muy bajo de tus usuarios.

Generar automáticamente la versión de texto

Probemos algo: en `TripController::show()`, elimina la línea `->textTemplate()`. Nuestro correo electrónico ahora sólo tiene versión HTML.

Haz otro viaje y comprueba el correo electrónico en Mailtrap. ¿Todavía tenemos una versión de texto? Se parece casi a nuestra plantilla de texto, pero con algún espaciado extra. Si envías un correo electrónico sólo con una versión HTML, Symfony Mailer crea automáticamente una versión de texto pero elimina las etiquetas. Es una buena alternativa, pero no es perfecta. ¿Ves lo que falta? El enlace Eso es... algo crítico... El enlace ha desaparecido porque estaba en el atributo `href` de la etiqueta de anclaje. Lo perdimos al eliminar las etiquetas.

Entonces, ¿necesitamos mantener siempre manualmente una versión de texto? No necesariamente. Aquí tienes un pequeño truco.

De HTML a Markdown

En tu terminal, ejecuta:

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

Este es un paquete que convierte HTML a markdown. Espera, ¿qué? ¿No solemos convertir markdown a HTML? Sí, pero para los correos electrónicos HTML, ¡esto es perfecto! ¿Y adivina qué? ¡No tenemos que hacer nada más! ¡Symfony Mailer utiliza automáticamente este paquete en lugar de limitarse a eliminar las etiquetas si están disponibles!

Reserva otro viaje y comprueba el correo electrónico en Mailtrap. El HTML parece el mismo, pero comprueba la versión de texto. ¡Nuestra etiqueta de anclaje se ha convertido en un enlace markdown! Todavía no es perfecto, ¡pero al menos está ahí! Si necesitas un control total, necesitarás esa plantilla de texto aparte, pero creo que esto es suficiente. De vuelta en tu IDE, borra `booking_confirmation.txt.twig`.

A continuación, ¡avivaremos este HTML con CSS!

Chapter 6: CSS en el correo electrónico

El CSS en el correo electrónico requiere... cierto cuidado especial. Pero, ¡pffff, somos desarrolladores de Symfony! ¡Avancemos temerariamente y veamos qué pasa!

Añade una clase CSS

En `email/booking_confirmation.html.twig`, añade una etiqueta `<style>` en `<head>` y añade una clase `.text-red` que establezca `color` en `red`:

templates/email/booking_confirmation.html.twig

```
<html>
<head>
    <style>
        .text-red {
            color: red;
        }
    </style>
</head>
// ... lines 9 - 26
</html>
```

Ahora, añade esta clase a la primera etiqueta `<p>`:

templates/email/booking_confirmation.html.twig

```
// ... lines 1 - 8
<body>
<p class="text-red">Hey {{ customer.name|split(' ')|first }},</p>
// ... lines 11 - 25
</body>
// ... lines 27 - 28
```

En nuestra aplicación, reserva otro viaje para nuestro buen amigo Steve. ¡Realmente está acumulando parsecs! ¿Crees que le interesaría la tarjeta de crédito platino Universal Travel?

En Mailtrap, comprueba el correo electrónico. Vale, este texto está en rojo como esperábamos... entonces, ¿cuál es el problema? Comprueba el código HTML para obtener una pista. Pasa el ratón por encima del primer error:

> *"La etiqueta `style` no es compatible con todos los clientes de correo electrónico."*

El problema más importante es el atributo `class`: tampoco es compatible con todos los clientes de correo electrónico. ¿Podemos viajar al espacio pero no podemos utilizar clases CSS en los correos electrónicos? Sí, es un mundo extraño.

CSS en línea

¿La solución? Haz como si estuviéramos en 1999 e inlinea todos los estilos. Así es, por cada etiqueta que tenga un `class`, tenemos que encontrar todos los estilos aplicados de la clase y añadirlos como atributo `style`. Manualmente, esto sería suuuuuck... Por suerte, ¡Symfony Mailer te tiene cubierto!

`inline_css` Filtro Twig

En la parte superior de este archivo, añade una etiqueta Twig `apply` con el filtro `inline_css`. Si no estás familiarizado, la etiqueta `apply` te permite aplicar cualquier filtro Twig a un bloque de contenido. Al final del archivo, escribe `endapply`:

templates/email/booking_confirmation.html.twig

```
{% apply inline_css %}
<html>
// ... lines 3 - 27
</html>
{% endapply %}
```

Reserva otro viaje para Steve. Uy, ¡un error! El filtro `inline_css` forma parte de un paquete que no tenemos instalado, ¡pero el mensaje de error nos da el comando `composer require` para instalarlo! Cópialo, salta a tu terminal y pégalo:

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

De vuelta en la aplicación, vuelve a reservar el viaje de Steve y comprueba el correo electrónico en Mailtrap.

El HTML parece el mismo, pero comprueba la Fuente HTML. ¡Este atributo `style` se añadió automáticamente a la etiqueta `<p>`! Es increíble y mucho mejor que hacerlo manualmente.

Si tu aplicación envía varios correos electrónicos, querrás que tengan un estilo coherente a partir de un archivo CSS real, en lugar de definirlo todo en una etiqueta `<style>` en cada plantilla. Por desgracia, no es tan sencillo como enlazar a un archivo CSS en la etiqueta `<head>`. Eso es algo que tampoco gusta a los clientes de correo electrónico.

¡No hay problema!

Archivo CSS externo

Crea un nuevo archivo `email.css` en `assets/styles/`. Copia el CSS de la plantilla de correo electrónico y pégalo aquí:

assets/styles/email.css

```
.text-red {
    color: red;
}
```

De vuelta en la plantilla, celébralo eliminando la etiqueta `<style>`.

Entonces, ¿cómo podemos hacer que nuestro correo electrónico utilice el archivo CSS externo? ¡Con trucos, por supuesto!

Espacio de nombres "styles" de Twig

Abre `config/packages/twig.yaml` y crea una clave `paths`. Dentro, añade `%kernel.project_dir%/assets/styles: styles`:

config/packages/twig.yaml

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

Lo sé, esto parece raro, pero crea un espacio de nombres Twig personalizado. Gracias a esto ahora podemos renderizar plantillas dentro de este directorio con el prefijo `@styles/`. Pero, ¡espera un momento! `email.css` ¡el archivo no es una plantilla Twig que queramos renderizar! No pasa nada, sólo necesitamos acceder a ella, no parsearla como Twig.

`inline_css()` con `source()`

De vuelta en `booking_confirmation.html.twig`, para el argumento de `inline_css`, utiliza `source('@styles/email.css')`:

templates/email/booking_confirmation.html.twig

```
{% apply inline_css(source('@styles/email.css')) %}
// ... lines 2 - 24
```

La función `source()` toma el contenido en bruto de un archivo.

Salta a nuestra aplicación, reserva otro viaje y comprueba el correo electrónico en Mailtrap. ¡Parece el mismo! Aquí el texto es rojo. Si comprobamos el código fuente HTML, las clases ya no están en `<head>`, pero los estilos siguen alineados: se están cargando desde nuestra hoja de estilos externa, ¡es genial!

A continuación, vamos a mejorar el HTML y el CSS para que este correo electrónico sea digno de la bandeja de entrada de Steve y del costoso viaje que acaba de reservar.

Chapter 7: Estilo de correo electrónico real con Inky y Foundation CSS

Para que este correo electrónico tenga un aspecto realmente elegante, tenemos que mejorar el HTML y el CSS.

Empecemos por el CSS. Con el CSS estándar de un sitio web, es probable que hayas utilizado un framework CSS como Tailwind (que utiliza nuestra aplicación), Bootstrap o Foundation. ¿Existe algo así para los correos electrónicos? Sí Y es aún más importante utilizar uno para los correos electrónicos porque hay muchos clientes de correo electrónico que los renderizan de forma diferente.

CSS de Foundation para correos electrónicos

Para los correos electrónicos, recomendamos utilizar Foundation, ya que tiene un marco específico para correos electrónicos. Busca en Google "Foundation CSS" y encontrarás esta página.

Descarga el kit de inicio para la "Versión CSS". Este archivo zip incluye un archivo `foundation-emails.css` que es el "framework" real.

Ya lo he incluido en el directorio `tutorials/`. Cópialo en `assets/styles/`.

En nuestro `booking_confirmation.html.twig`, el filtro `inline_css` puede tomar varios argumentos. Haz que el primer argumento sea `source('@styles/foundation-emails.css')` y utiliza `email.css` para el segundo argumento:

templates/email/booking_confirmation.html.twig

```
{% apply inline_css(source('@styles/foundation-emails.css'), source('@styles/email.css')) %}
// ... lines 2 - 24
```

Esto contendrá estilos personalizados y anulaciones.

Abriré `email.css` y pegaré algo de CSS personalizado para nuestro correo electrónico:

assets/styles/email.css

```
.trip-name {
    font-size: 32px;
}

.accent-title {
    color: #666666;
}

.trip-image {
    border-radius: 12px;
}
```

¡Tablas!

Ahora tenemos que mejorar nuestro HTML. Pero ¡qué noticia más rara! La mayoría de las cosas que utilizamos para dar estilo a los sitios web no funcionan en los correos electrónicos. Por ejemplo, no podemos utilizar Flexbox ni Grid. En su lugar, tenemos que utilizar tablas para la maquetación. ¡Tablas! Tablas, dentro de tablas, dentro de tablas. ¡Qué asco!

Lenguaje de plantillas Inky

Por suerte, hay un lenguaje de plantillas que podemos utilizar para hacer esto más fácil. Busca "inky templating language" para encontrar esta página. Inky está desarrollado por la Fundación Zurb. Zurb, Inky, Foundation... ¡estos nombres encajan perfectamente con nuestro tema espacial! ¡Y todos funcionan juntos!

Puedes hacerte una idea de cómo funciona en la vista general. Este es el HTML necesario para un simple correo electrónico. ¡Es un infierno de tabla! Haz clic en la pestaña "Cambiar a Inky". ¡Guau! ¡Esto es mucho más limpio! Escribimos en un formato más legible e Inky lo convierte en la tabla-horror necesaria para los correos electrónicos.

Incluso hay "componentes Inky": botones, llamadas, cuadrículas, etc.

En tu terminal, instala un filtro Twig de Inky que convertirá nuestro marcado Inky en HTML.

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

`inky_to_html` Filtro Twig

En `booking_confirmation.html.twig`, añade el filtro `inky_to_html` a `apply`, canalizando `inline_css` a continuación:

templates/email/booking_confirmation.html.twig

```
{% apply inky_to_html|inline_css(source('@styles/foundation-emails.css'), source('@styles/email.css')) %}
// ... lines 2 - 24
```

En primer lugar, aplicamos el filtro Inky y, a continuación, alineamos el CSS.

Copiaré algunas marcas Inky para nuestro correo electrónico.

templates/email/booking_confirmation.html.twig

```
{% apply inky_to_html|inline_css(source('@styles/foundation-emails.css'), source('@styles/email.css')) %}
    <container>
        <row>
            <columns>
                <spacer size="40"></spacer>
                <p class="accent-title">Get Ready for your trip to</p>
                <h1 class="trip-name">{{ trip.name }}</h1>
            </columns>
        </row>
        <row>
            <columns>
                <p class="accent-title">Departure: {{ booking.date|date('Y-m-d') }}</p>
            </columns>
        </row>
        <row>
            <columns>
                <button class="expanded rounded center" href="{{ url('booking_show', {uid: booking.uid}) }}">
                    Manage Booking
                </button>
                <button class="expanded rounded center secondary" href="{{ url('bookings', {uid: customer.uid}) }}">
                    My Account
                </button>
            </columns>
        </row>
        <row>
            <columns>
                <p>We can't wait to see you there,</p>
                <p>Your friends at Universal Travel</p>
            </columns>
        </row>
    </container>
{% endapply %}
```

Tenemos un `<container>`, con `<rows>` y `<columns>`. Este será un correo electrónico de una sola columna, pero puedes tener tantas columnas como necesites. Este `<spacer>` añade espacio vertical para respirar.

¡Veamos este correo electrónico en acción! Reserva un nuevo viaje para Steve, ¡ups, debe ser una fecha en el futuro, y reserva!

Comprueba Mailtrap y encuentra el correo electrónico. ¡Vaya! ¡Esto tiene mucho mejor aspecto! Podemos utilizar este pequeño widget que Mailtrap proporciona para ver cómo se verá en móviles y tabletas.

Mirando el "HTML Check", parece que tenemos algunos problemas, pero, creo que mientras estemos usando Foundation e Inky como es debido, deberíamos estar bien.

Comprueba los botones. "Gestionar reserva", sí, funciona. "Mi cuenta", sí, también funciona. ¡Eso ha sido un éxito rápido gracias a Foundation e Inky!

A continuación, vamos a mejorar aún más nuestro correo electrónico incrustando la imagen del viaje y haciendo felices a los abogados añadiendo un archivo adjunto en PDF con las "condiciones del servicio".

Chapter 8: Archivos adjuntos e imágenes

¿Podemos añadir un archivo adjunto a nuestro correo electrónico? Por supuesto que sí Hacerlo manualmente es un proceso complejo y delicado. Por suerte, el Mailer de Symfony te lo pone muy fácil.

En el directorio `tutorial/`, verás un archivo `terms-of-service.pdf`. Muévelo a `assets/`, aunque podría estar en cualquier sitio.

En `TripController::show()`, necesitamos obtener la ruta a este archivo. Añade un nuevo argumento `string $termsPath` y con el atributo `#[Autowire]` y `%kernel.project_dir%/assets/terms-of-service.pdf'`:

src/Controller/TripController.php

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

Genial, ¿verdad?

Adjunta

Abajo, donde creamos el correo electrónico, escribe `->attach` y mira lo que te sugiere tu IDE. Hay dos métodos: `attach()` y `attachFromPath()`. `attach()` es para añadir el contenido en bruto de un archivo (como cadena o flujo). Como nuestro adjunto es un archivo real en nuestro sistema de archivos, utiliza `attachFromPath()` y pasa `$termsPath` y luego un nombre amigable como `Terms of Service.pdf`:

src/Controller/TripController.php

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

Este será el nombre del archivo cuando se descargue. Si no se pasa el segundo argumento, por defecto será el nombre del archivo.

Adjunto hecho. ¡Ha sido fácil!

Incrustar imágenes

A continuación, vamos a añadir la imagen del viaje al correo electrónico de confirmación de la reserva. Pero no la queremos como archivo adjunto. La queremos incrustada en el HTML. Hay dos formas de hacerlo: Primero, la forma estándar de la web: utilizar una etiqueta `<img>` con una URL absoluta a la imagen alojada en tu sitio. Pero vamos a ser inteligentes e incrustar la imagen directamente en el correo electrónico. Esto es como un archivo adjunto, pero no está disponible para su descarga, sino que haces referencia a ella en el HTML de tu correo electrónico.

Primero, como hicimos con nuestros archivos CSS externos, tenemos que hacer que nuestras imágenes estén disponibles en Twig. `public/imgs/` contiene las imágenes de nuestro viaje y todas se llaman `<trip-slug.png>`.

En `config/packages/twig.yaml`, añade otra entrada `paths`: `%kernel.project_dir%/public/imgs: images`:

config/packages/twig.yaml

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

Ahora podemos acceder a este directorio en Twig con `@images/`. Cierra este archivo.

La variable `email`

Cuando utilizas Twig para procesar tus correos electrónicos, por supuesto tienes acceso a las variables pasadas a `->context()` pero también hay una variable secreta disponible llamada `email`. Ésta es una instancia de `WrappedTemplatedEmail` y te da acceso a cosas relacionadas con el correo electrónico como el asunto, la ruta de retorno, de, a, etc. Lo que nos interesa es este método `image()`. ¡Es el que se encarga de incrustar imágenes!

¡Vamos a utilizarlo!

En `booking_confirmation.html.twig`, debajo de este `<h1>`, añade una etiqueta `<img>` con algunas clases: `trip-image` de nuestro archivo CSS personalizado y `float-center` de Foundation.

Para el `src`, escribe `{{ email.image() }}`, este es el método de ese objeto `WrappedTemplatedEmail`. Dentro, escribe `'@images/%s.png'|format(trip.slug)`. Añade un `alt="{{ trip.name }}"` y cierra la etiqueta:

templates/email/booking_confirmation.html.twig

```
{% apply inky_to_html|inline_css(source('@styles/foundation-emails.css'), source('@styles/email.css')) %}
    <container>
        <row>
            <columns>
// ... lines 5 - 6
                <h1 class="trip-name">{{ trip.name }}</h1>
                <img
                    class="trip-image float-center"
                    src="{{ email.image('@images/%s.png'|format(trip.slug)) }}"
                    alt="{{ trip.name }}">
            </columns>
        </row>
// ... lines 14 - 34
    </container>
{% endapply %}
```

¡Imagen incrustada! ¡Vamos a comprobarlo!

De vuelta en la aplicación, reserva un viaje... y comprueba Mailtrap. Aquí está nuestro correo electrónico y... ¡aquí está nuestra imagen! ¡Somos lo máximo! Encaja perfectamente e incluso tiene unas bonitas esquinas redondeadas.

Aquí arriba, en la parte superior derecha, vemos "Adjunto (1)", tal y como esperábamos. Haz clic en él y elige "Condiciones de servicio.pdf" para descargarlo. Ábrelo y... ¡ahí está nuestro PDF! Nuestros abogados espaciales han hecho divertido este documento, ¡y sólo nos ha costado 500 créditos/hora! ¡Créditos de inversor bien invertidos!

A continuación, vamos a eliminar la necesidad de poner manualmente un `from` a cada correo electrónico, utilizando eventos para añadirlo globalmente.

Chapter 9: Global Desde (y Diversión) con Eventos de Correo Electrónico

Apuesto a que la mayoría, si no todos, los correos electrónicos que envíe tu aplicación tendrán la misma dirección de correo electrónico, algo ingenioso como `hal9000@universal-travel.com` o el probado pero más soporífero `info@universal-travel.com`.

Como todos los correos tendrán la misma dirección de origen, no tiene sentido establecerla en todos los correos. Curiosamente, no hay ninguna opción de configuración minúscula para esto. Pero eso es genial para nosotros: ¡nos da la oportunidad de aprender sobre eventos! Muy potente, muy friki.

El `MessageEvent`

Antes de enviar un correo electrónico, Mailer envía un mensaje `MessageEvent`.

Para escucharlo, busca tu terminal y ejecuta:

```
symfony console make:listener
```

Llámalo `GlobalFromEmailListener`. El nos da una lista de eventos que podemos escuchar. Queremos el primero: `MessageEvent`. Empieza a escribir `Symfony` y se autocompletará por nosotros. Pulsa intro.

¡Escucha creada!

Para ser más guays, pongamos nuestra dirección global de origen como parámetro. En `config/services.yaml`, debajo de `parameters`, añade una nueva: `global_from_email`.

Cadena especial de dirección de correo electrónico

Esto será una cadena, pero fíjate en esto: ponlo en `Universal Travel` , luego entre paréntesis angulares, pon el correo electrónico: `<info@universal-travel.com>`:

config/services.yaml

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

Cuando Symfony Mailer vea una cadena con este aspecto como dirección de correo electrónico, creará el objeto `Address` adecuado con un nombre y un correo electrónico establecidos. ¡Genial!

`MessageEvent` Receptor

Abre la nueva clase `src/EventListener/GlobalFromEmailListener.php`. Añade un constructor con un argumento `private string $fromEmail` y un atributo `#[Autowire]` con el nombre de nuestro parámetro: `%global_from_email%`:

src/EventListener/GlobalFromEmailListener.php

```
// ... lines 1 - 8
final class GlobalFromEmailListener
{
    public function __construct(
        #[Autowire('%global_from_email%')]
        private string $fromEmail,
    ) {
    }
// ... lines 16 - 21
}
```

Aquí abajo, el atributo `#[AsEventListener]` es lo que marca este método como un oyente de eventos. En realidad, podemos eliminar este argumento `event` - se deducirá de la sugerencia de tipo del argumento del método: `MessageEvent`:

src/EventListener/GlobalFromEmailListener.php

```
// ... lines 1 - 9
final class GlobalFromEmailListener
{
// ... lines 12 - 17
    #[AsEventListener]
    public function onMessageEvent(MessageEvent $event): void
    {
// ... lines 21 - 31
    }
}
```

Dentro, primero coge el mensaje del evento: `$message = $event->getMessage()`:

src/EventListener/GlobalFromEmailListener.php

```
// ... lines 1 - 9
final class GlobalFromEmailListener
{
// ... lines 12 - 18
    public function onMessageEvent(MessageEvent $event): void
    {
        $message = $event->getMessage();
// ... lines 22 - 31
    }
}
```

Salta al método `getMessage()` para ver lo que devuelve. `RawMessage`... salta a esto y mira qué clases lo extienden. `TemplatedEmail` ¡! ¡Perfecto!

De vuelta a nuestro oyente, escribe `if (!$message instanceof TemplatedEmail)`, y dentro, `return;`:

src/EventListener/GlobalFromEmailListener.php

```
// ... lines 1 - 9
final class GlobalFromEmailListener
{
// ... lines 12 - 18
    public function onMessageEvent(MessageEvent $event): void
    {
// ... lines 21 - 22
        if (!$message instanceof TemplatedEmail) {
            return;
        }
// ... lines 26 - 31
    }
}
```

Es probable que esto no ocurra nunca, pero es una buena práctica volver a comprobarlo. Además, ayuda a nuestro IDE a saber que `$message` es ahora un `TemplatedEmail`.

Es posible que un correo electrónico aún establezca su propia dirección `from`. En este caso, no queremos anularla. Así que añade una cláusula de protección `if ($message->getFrom())`, `return;`:

src/EventListener/GlobalFromEmailListener.php

```
// ... lines 1 - 9
final class GlobalFromEmailListener
{
// ... lines 12 - 18
    public function onMessageEvent(MessageEvent $event): void
    {
// ... lines 21 - 26
        if ($message->getFrom()) {
            return;
        }
// ... lines 30 - 31
    }
}
```

Ahora, podemos establecer la global `from`: `$message->from($this->fromEmail)`:

src/EventListener/GlobalFromEmailListener.php

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

¡Perfecto!

De vuelta en `TripController::show()`, elimina el `->from()` para el correo electrónico.

¡Es hora de probarlo! En nuestra aplicación, reserva un viaje y comprueba Mailtrap para el correo electrónico. Redoble de tambores... ¡el `from` está configurado correctamente! ¡Nuestro oyente funciona! Nunca dudé de nosotros.

`Reply-To`

Un detalle más para que esto sea completamente hermético (como la mayoría de nuestros barcos).

Imagina un formulario de contacto en el que el usuario rellena su nombre, correo electrónico y un mensaje. Esto lanza un correo electrónico con estos datos a tu equipo de soporte. En sus clientes de correo electrónico, estaría bien que, cuando pulsen responder, vaya al correo del formulario, no a tu "global de".

Podrías pensar que deberías establecer la dirección `from` en el correo electrónico del usuario, pero eso no funcionará, ya que no estamos autorizados a enviar correos electrónicos en nombre de ese usuario. Pronto hablaremos más sobre la seguridad del correo electrónico.

Afortunadamente, existe una cabecera de correo electrónico especial llamada `Reply-To` precisamente para este escenario. Cuando construyas tu correo electrónico, configúrala con `->replyTo()` y pasa la dirección de correo electrónico del usuario.

Abróchate el cinturón porque los tanques de refuerzo están llenos y listos para el lanzamiento! Es hora de enviar correos electrónicos reales en producción! Eso a continuación.

Chapter 10: Envío en producción con Mailtrap

Muy bien, ¡por fin ha llegado el momento de enviar correos electrónicos reales en producción!

Transportes de Mailer

Mailer viene con varias formas de enviar correos electrónicos, llamadas "transportes". Este `smtp` es el que estamos utilizando para nuestras pruebas con Mailtrap. Podríamos configurar nuestro propio servidor SMTP para enviar correos... pero... eso es complejo, y tienes que hacer un montón de cosas para asegurarte de que tus correos no se marcan como spam. Boo.

transportes de terceros

Te recomiendo encarecidamente que utilices un servicio de correo electrónico de terceros. Éstos gestionan todas estas complejidades por ti y Mailer proporciona puentes a muchos de ellos para que la configuración sea pan comido.

Puente Mailtrap

Utilizamos Mailtrap para las pruebas, pero Mailtrap también tiene funciones de envío a producción ¡Fantástico! Incluso tiene un puente oficial

En tu terminal, instálalo con:

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

Una vez instalado, comprueba tu IDE. En `.env`, la receta añade algunos stubs de `MAILER_DSN`. Podemos obtener los valores DSN reales de Mailtrap, pero antes tenemos que hacer algunos ajustes.

Dominio de envío

En Mailtrap, tenemos que configurar un "dominio de envío". Esto configura un dominio de tu propiedad para permitir que Mailtrap envíe correos electrónicos correctamente en su nombre.

Nuestros abogados aún están negociando la compra de `universal-travel.com`, así que, por ahora, estoy utilizando un dominio personal que poseo: `zenstruck.com`. Añade tu dominio aquí.

Una vez añadido, estarás en esta página de "Verificación del dominio". Esto es súper importante, pero Mailtrap lo hace fácil. Sólo tienes que seguir las instrucciones hasta que aparezca esta marca de verificación verde. Básicamente, tendrás que añadir un montón de registros DNS específicos a tu dominio. DKIM, que verifica los correos electrónicos enviados desde tu dominio, y SPF, que autoriza a Mailtrap a enviar correos electrónicos en nombre de tu dominio, son los más importantes. Mailtrap proporciona una gran documentación sobre ellos si quieres profundizar en cómo funcionan exactamente. Pero básicamente, le estamos diciendo al mundo que Mailtrap está autorizado a enviar correos electrónicos en nuestro nombre.

Producción `MAILER_DSN`

Una vez que tengas la marca de verificación verde, haz clic en "Integraciones" y luego en "Integrar" en la sección "Flujo de transacciones".

Ahora podemos decidir entre utilizar SMTP o API. Yo utilizaré la API, pero cualquiera de las dos funciona. Y ¡hey! Esto me resulta familiar: como con las pruebas de Mailtrap, elige PHP y luego Symfony. ¡Este es el `MAILER_DSN` que necesitamos! Cópialo y salta a tu editor.

Se trata de una variable de entorno sensible, así que añádela a `.env.local` para evitar confirmarla en git. Comenta el DSN de prueba de Mailtrap y pégalo a continuación. Eliminaré este comentario porque nos gusta mantener la vida ordenada.

¡Casi listo! Recuerda que sólo podemos enviar correos en producción desde el dominio que hemos configurado. En mi caso, `zenstruck.com`. Abre `config/services.yaml` y actualiza el `global_from_email` a tu dominio.

¡Veamos si funciona! En tu aplicación, reserva un viaje. Esta vez utiliza una dirección de correo electrónico real. Pondré el nombre `Kevin` y utilizaré mi correo electrónico personal: `kevin@symfonycasts.com`. Por mucho que te quiera a ti y a los viajes espaciales, pon aquí tu propio correo electrónico para evitar enviarme spam. ¡Elige una fecha y reserva!

Estamos en la página de confirmación de la reserva, ¡es una buena señal! Ahora, comprueba tu correo electrónico personal. Yo voy al mío y espero... actualizo... ¡aquí está! Si hago clic, ¡esto es exactamente lo que esperamos! La imagen, el archivo adjunto, ¡todo está aquí!

A continuación, vamos a ver cómo podemos rastrear los correos electrónicos enviados con Mailtrap, ¡además de añadir etiquetas y metadatos para mejorar ese rastreo!

Chapter 11: Seguimiento de correos electrónicos con etiquetas y metadatos

Ya estamos enviando correos electrónicos de verdad. Comprobemos que nuestros enlaces funcionan... ¡Todo bien!

Registros de correo electrónico Mailtrap

Mailtrap puede hacer algo más que enviar y depurar correos electrónicos: también podemos rastrear correos electrónicos y eventos de correo electrónico. Entra en Mailtrap y haz clic en "Email API/SMTP". Este panel nos muestra un resumen de cada correo electrónico que hemos enviado. Haz clic en "Registros de correo electrónico" para ver la lista completa. ¡Aquí está nuestro correo electrónico! Haz clic en él para ver los detalles.

Esto te resulta familiar... es similar a la interfaz de pruebas de Mailtrap. Podemos ver detalles generales, un análisis de spam y mucho más. Pero esto es realmente genial: haz clic en "Historial de Eventos". Esto muestra todos los eventos que ocurrieron durante el flujo de este correo electrónico. Podemos ver cuándo se envió, cuándo se entregó, ¡incluso cuándo lo abrió el destinatario! Cada evento tiene detalles adicionales, como la dirección IP que abrió el correo electrónico. Súper útil para diagnosticar problemas de correo electrónico. Mailtrap también tiene una función de seguimiento de enlaces que, si está activada, mostraría qué enlaces se pulsaron en el correo electrónico.

De vuelta a la pestaña "Información del correo electrónico", desplázate un poco hacia abajo. Observa que falta la "Categoría". En realidad, esto no es un problema, pero una "categoría" es una cadena que ayuda a organizar los distintos correos electrónicos que envía tu aplicación. Esto facilita la búsqueda y puede darnos estadísticas interesantes como "¿cuántos correos electrónicos de registro de usuarios enviamos el mes pasado?".

Etiqueta de correo electrónico (categoría Mailtrap)

Symfony Mailer llama a esto una "etiqueta" que puedes añadir a los correos electrónicos. El puente Mailtrap toma esta etiqueta y la convierte en su "categoría". ¡Vamos a añadir una!

En `TripController::show()`, después de la creación del correo electrónico, escribe: `$email->getHeaders()->add(new TagHeader());` - utiliza `booking` como nombre:

src/Controller/TripController.php

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

Metadatos del correo electrónico (Variables personalizadas de Mailtrap)

Mailer también tiene una cabecera especial de metadatos que puedes añadir a los correos electrónicos. Se trata de un almacén clave-valor de forma libre para añadir datos adicionales. El puente Mailtrap los convierte en lo que ellos llaman "variables personalizadas".

Vamos a añadir un par:

src/Controller/TripController.php

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

Y:

src/Controller/TripController.php

```
// ... lines 1 - 22
final class TripController extends AbstractController
{
// ... lines 25 - 33
    public function show(
// ... lines 35 - 42
    ): Response {
// ... lines 44 - 45
        if ($form->isSubmitted() && $form->isValid()) {
// ... lines 47 - 69
            $email->getHeaders()->add(new MetadataHeader('customer_uid', $customer->getUid()));
// ... lines 71 - 74
        }
// ... lines 76 - 80
    }
}
```

A cada correo electrónico de reserva se adjunta ahora una referencia al cliente y a la reserva. ¡Fantástico!

Para ver cómo se verán en Mailtrap, salta a nuestra aplicación y reserva un viaje (recuerda que aún estamos utilizando el envío de producción, así que utiliza tu correo electrónico personal). Comprueba nuestra bandeja de entrada... aquí está. De vuelta en Mailtrap, vuelve a los registros de correo electrónico... y actualiza... ¡ahí está! Haz clic en él. Ahora, en esta pestaña "Información de correo electrónico", ¡vemos nuestra categoría "reserva"! Un poco más abajo, están nuestros metadatos o "variables personalizadas".

Filtrar por categoría

Para filtrar por "categoría", ve a los registros de correo electrónico. En este cuadro de búsqueda, elige "Categorías". Este filtro enumera todas las categorías que hemos utilizado. Selecciona "reserva" y "Buscar". Esto ya está más organizado que los tubos Jeffries de ingeniería

¡Esto es el envío de correos electrónicos de producción con Mailtrap! Para facilitar las cosas en los próximos capítulos, volvamos a utilizar Mailtrap en pruebas. En `.env.local`, descomenta la prueba de Mailtrap `MAILER_DSN` y comenta el envío de producción `MAILER_DSN`.

A continuación, vamos a utilizar Symfony Messenger para enviar nuestros correos electrónicos de forma asíncrona. ¡Ooo!

Chapter 12: Envío asíncrono y reintentable con Messenger

Cuando enviamos este correo electrónico, se envía inmediatamente, de forma sincrónica. Esto significa que nuestro usuario ve un retraso mientras nos conectamos al transporte de correo para enviar el correo electrónico. Y si hay un problema de red por el que el correo falla, el usuario verá un error 500: no inspira precisamente confianza en una empresa que va a atarte a un cohete.

En lugar de eso, enviemos nuestros correos electrónicos de forma asíncrona. Esto significa que, durante la petición, el correo electrónico se enviará a una cola para ser procesado más tarde. ¡Symfony Messenger es perfecto para esto! Y obtenemos las siguientes ventajas: respuestas más rápidas para el usuario, reintentos automáticos si el correo electrónico falla, y la posibilidad de marcar los correos electrónicos para su revisión manual si fallan demasiadas veces.

Instalación de Messenger y Doctrine Transport

¡Vamos a instalar Messenger! En tu terminal, ejecuta:

```
composer require messenger
```

Al igual que Mailer, Messenger tiene el concepto de transporte: aquí es donde se envían los mensajes para ponerlos en cola. Utilizaremos el transporte Doctrine, ya que es el más fácil de configurar.

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

En nuestro IDE, la receta añadía este `MESSENGER_TRANSPORT_DSN` a nuestro `.env` y por defecto era Doctrine: ¡perfecto! Este transporte añade una tabla a nuestra base de datos, así que técnicamente deberíamos crear una migración para ello. Pero... vamos a hacer un poco de trampa y hacer que cree automáticamente la tabla si no existe. Para permitirlo, configura `auto_setup` en `1`:

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

Configurar los transportes de Messenger

La receta también ha creado este archivo `config/packages/messenger.yaml`. Descomenta la línea `failure_transport`:

```
config/packages/messenger.yaml
1 framework:
2     messenger:
↕ // ... line 3
4         failure_transport: failed
↕ // ... lines 5 - 24
```

Esto activa el sistema de revisión manual de fallos que he mencionado antes. A continuación, descomenta la línea `async` debajo de `transports`:

config/packages/messenger.yaml

```
framework:
    messenger:
// ... lines 3 - 5
        transports:
// ... line 7
            async: '%env(MESSENGER_TRANSPORT_DSN)%'
// ... lines 9 - 24
```

Esto habilita el transporte configurado con `MESSENGER_TRANSPORT_DSN` y lo nombra `async`. No es obvio aquí, pero los mensajes fallidos se vuelven a intentar 3 veces, con un retraso creciente entre cada intento. Si un mensaje sigue fallando después de 3 intentos, se envía a `failure_transport`, llamado `failed`, así que descomenta también este transporte:

config/packages/messenger.yaml

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

Configurar el enrutamiento de Messenger

La sección `routing` es donde le decimos a Symfony qué mensajes deben enviarse a qué transporte. Mailer utiliza una clase de mensaje específica para enviar correos electrónicos. Así que envía `Symfony\Component\Mailer\Messenger\SendEmailMessage` al transporte `async`:

config/packages/messenger.yaml

```
framework:
    messenger:
// ... lines 3 - 11
        routing:
// ... lines 13 - 14
            'Symfony\Component\Mailer\Messenger\SendEmailMessage': async
// ... lines 16 - 24
```

¡Ya está! Symfony Messenger y Mailer se acoplan perfectamente, así que no tenemos que cambiar nada en nuestro código.

¡Vamos a probarlo! De vuelta en nuestra aplicación... reserva un viaje. Volvemos a utilizar el transporte de pruebas de Mailtrap, así que podemos utilizar cualquier correo electrónico. Ahora observa cuánto más rápido se procesa.

¡Bum!

Estado: En cola

Abre el perfil de la última petición y comprueba la sección "Correos electrónicos". Parece normal, pero fíjate en que el Estado es "En cola". Se envió a nuestro transporte Messenger, no a nuestro transporte Mailer. Tenemos esta nueva sección "Mensajes". Aquí podemos ver el `SendEmailMessage` que contiene nuestro objeto `TemplatedEmail`.

Salta a Mailtrap y actualiza... todavía nada. ¡Por supuesto! Tenemos que procesar nuestra cola.

Procesar la cola

Vuelve a tu terminal y ejecuta:

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

Esto procesa nuestro transporte `async` (el `-vv` sólo añade más salida para que podamos ver lo que ocurre). ¡Muy bien! El mensaje se ha recibido y gestionado correctamente. Es decir: esto debería haber enviado realmente el correo electrónico.

Comprueba Mailtrap... ¡ya está aquí! Parece correcto... pero... haz clic en uno de nuestros enlaces.

¿Pero qué? Comprueba la URL: ¡es el dominio equivocado! Averigüemos qué parte de nuestro cohete de correo electrónico ha causado esto y arreglémoslo a continuación

Chapter 13: Generar URLs en el entorno CLI

Cuando cambiamos al envío asíncrono de correo electrónico, ¡rompimos nuestros enlaces de correo electrónico! Está utilizando `localhost` como nuestro dominio, raro e incorrecto.

De vuelta en nuestra aplicación, podemos obtener una pista de lo que está pasando mirando el perfil de la petición que envió el correo electrónico. Recuerda que ahora nuestro correo electrónico está marcado como "en cola". Ve a la pestaña "Mensajes" y busca el mensaje: `SendEmailMessage`. Dentro está el objeto `TemplatedEmail`. Ábrelo. Interesante! `htmlTemplate` es nuestra plantilla Twig pero `html` es `null`! ¿No debería ser el HTML renderizado de esa plantilla? Este pequeño detalle es importante: la plantilla de correo electrónico no se renderiza cuando nuestro controlador envía el mensaje a la cola. No! la plantilla no se renderiza hasta más tarde, cuando ejecutamos `messenger:consume`.

Generación de enlaces en la CLI

¿Qué importancia tiene esto? Bueno `messenger:consume` es un comando CLI, y cuando se generan URLs absolutas en la CLI, Symfony no sabe cuál debe ser el dominio (o si debe ser http o https). Entonces, ¿por qué lo hace cuando está en un controlador? En un controlador, Symfony utiliza la petición actual para averiguarlo. En un comando CLI, no hay petición, así que se rinde y utiliza `http://localhost`.

Configurar la URL por defecto

Vamos a decirle cuál debe ser el dominio.

De vuelta a nuestro IDE, abre `config/packages/routing.yaml`. En `framework`, `routing`, estos comentarios explican exactamente esta cuestión. Descomenta `default_uri` y ajústalo a `https://universal-travel.com` - ¡nuestros abogados están a punto de llegar a un acuerdo!

config/packages/routing.yaml

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

En desarrollo, sin embargo, tenemos que utilizar la URL de nuestro servidor local de desarrollo. Para mí, es `127.0.0.1:8000`, pero puede ser diferente para otros miembros del equipo. Sé que Bob utiliza `bob.is.awesome:8000` y más o menos es así.

URL predeterminada del entorno de desarrollo

Para que esto sea configurable, hay un truco: el servidor Symfony CLI establece una variable de entorno especial con el dominio llamado `SYMFONY_PROJECT_DEFAULT_ROUTE_URL`.

De vuelta en nuestra configuración de enrutamiento, añade una nueva sección: `when@dev:`, `framework:`, `router:`, `default_uri:` y establécela en `%env(SYMFONY_PROJECT_DEFAULT_ROUTE_URL)%`:

config/packages/routing.yaml

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

Esta variable de entorno sólo estará disponible si el servidor CLI de Symfony se está ejecutando y estás ejecutando comandos a través de `symfony console` (no `bin/console`). Para evitar un error si falta la variable, establece una por defecto. Todavía en `when@dev`, añade `parameters:` con `env(SYMFONY_PROJECT_DEFAULT_ROUTE_URL):` establecido en `http://localhost`.

config/packages/routing.yaml

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

Esta es la forma estándar de Symfony de establecer un valor por defecto para una variable de entorno.

Reinicia `messenger:consume`

¡Hora de probar! Pero primero, vuelve a tu terminal. Como hemos hecho algunos cambios en nuestra configuración, tenemos que reiniciar el comando `messenger:consume` para, más o menos, recargar nuestra aplicación:

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

¡Genial! El comando se ejecuta de nuevo y utiliza nuestra nueva configuración de Symfony. Vuelve a nuestra aplicación... ¡y reserva un viaje! Vuelve rápidamente al terminal... y veremos que el mensaje se ha procesado.

Ve a Mailtrap y... ¡aquí está! Momento de la verdad: haz clic en un enlace... Genial, ¡vuelve a funcionar! ¡Bob estará tan contento!

Ejecutando `messenger:consume` en segundo plano

Si eres como yo, probablemente te parezca un rollo tener que mantener este comando `messenger:consume` ejecutándose en un terminal durante el desarrollo. Además, tener que reiniciarlo cada vez que haces un cambio en el código o en la configuración es molesto. ¡Estoy harto! ¡Es hora de devolver la diversión a nuestras funciones con otro truco de la CLI de Symfony!

En tu IDE, abre este archivo `.symfony.local.yaml`. Es la configuración del servidor Symfony CLI para nuestra aplicación. ¿Ves esta clave `workers`? Nos permite definir procesos que se ejecutarán en segundo plano cuando iniciemos el servidor. Ya tenemos el comando tailwind configurado.

Añade otro trabajador. Llámalo `messenger` -aunque podría ser cualquier cosa- y establece `cmd` en `['symfony', 'console', 'messenger:consume', 'async']`:

.symfony.local.yaml

```
workers:
// ... lines 2 - 5
    messenger:
        cmd: ['symfony', 'console', 'messenger:consume', 'async']
// ... lines 8 - 9
```

Esto resuelve el problema de tener que mantenerlo en ejecución en una ventana de terminal independiente. Pero, ¿qué pasa con el reinicio del comando cuando hacemos cambios? No hay problema! Añade una clave `watch` y ajústala a `config`, `src`, `templates` y `vendor`:

.symfony.local.yaml

```
workers:
// ... lines 2 - 5
    messenger:
// ... line 7
        watch: ['config', 'src', 'templates', 'vendor']
```

Si cambia algún archivo de estos directorios, el trabajador se reiniciará solo ¡Inteligente!

De vuelta a tu terminal, reinicia el servidor con `symfony server:stop` y `symfony serve -d` `messenger:consume` ¡debería estar ejecutándose en segundo plano! Para comprobarlo, ejecuta

```
symfony server:status
```

¡3 trabajadores funcionando! El servidor web PHP real, el trabajador `tailwind:build` existente y nuestro nuevo `messenger:consume`. ¡Genial!

A continuación, ¡exploremos cómo hacer afirmaciones sobre correos electrónicos en nuestras pruebas funcionales!

Chapter 14: Aserciones de correos electrónicos en pruebas funcionales

Bien, ¡hora de hacer pruebas! Si has explorado un poco la base de código, te habrás dado cuenta de que alguien (podría haber sido cualquiera... pero probablemente un canadiense) coló algunas pruebas en nuestro directorio `tests/Functional/`. ¿Pasarán? Ni idea Averigüémoslo

Ve a tu terminal y ejecuta:

```
bin/phpunit
```

Uh-oh, 1 fallo. Uh-oh, porque, la verdad, ¡soy el simpático canadiense que las añadió y sé que pasaban al principio del curso! El fallo está en `BookingTest`, concretamente, `testCreateBooking`:

> *"Se esperaba un código de estado de redirección pero se obtuvo 500"*

en la línea 38 de `BookingTest`. Ahí es donde enviamos el correo electrónico... así que si buscamos a alguien a quien culpar, creo que deberíamos empezar por el canadiense, ejem, yo y mis salvajes maneras de enviar correos electrónicos.

Foundry y el navegador

Abre `BookingTest.php`. Si has escrito pruebas funcionales con Symfony antes, esto puede parecer un poco diferente porque estoy utilizando algunas bibliotecas de ayuda. `zenstruck/foundry` nos da este rasgo `ResetDatabase` que limpia la base de datos antes de cada prueba. También nos proporciona este rasgo `Factories` que nos permite crear fijaciones de base de datos en nuestras pruebas. Y `HasBrowser` es de otro paquete - `zenstruck/browser` - y es esencialmente una envoltura fácil de usar alrededor del cliente de pruebas de Symfony.

`testCreateBooking` es la prueba real. En primer lugar, creamos un `Trip` en la base de datos con estos valores conocidos. A continuación, algunas preaserciones para asegurarnos de que no hay reservas ni clientes en la base de datos. Ahora, utilizamos `->browser()` para navegar a la página de un viaje, rellenar el formulario de reserva y enviarlo. A continuación, afirmamos que se nos redirige a una URL de reserva específica y comprobamos que la página contiene algún HTML esperado. Por último, utilizamos Foundry para hacer algunas afirmaciones sobre los datos de nuestra base de datos.

`->throwExceptions()`

La línea 38 causó el fallo... estamos obteniendo un código de respuesta 500 al redirigir a esta página de reservas. los códigos de estado 500 en las pruebas pueden ser frustrantes porque puede ser difícil localizar la excepción real. Por suerte, Browser nos permite lanzar la excepción real. Al principio de esta cadena, añade `->throwExceptions()`:

tests/Functional/BookingTest.php

```
// ... lines 1 - 12
class BookingTest extends KernelTestCase
{
// ... lines 15 - 19
    public function testCreateBooking(): void
    {
// ... lines 22 - 30
        $this->browser()
            ->throwExceptions()
// ... lines 33 - 42
        ;
// ... lines 44 - 52
    }
}
```

De vuelta al terminal, vuelve a ejecutar las pruebas:

```
bin/phpunit
```

Ahora vemos una excepción No se puede encontrar la plantilla "@images/mars.png". Si recuerdas, esto se parece a cómo estamos incrustando las imágenes del viaje en nuestro correo electrónico. Está fallando porque `mars.png` no existe en `public/imgs`. Para simplificar, vamos a ajustar nuestra prueba para utilizar una imagen existente. Para nuestra fijación aquí, cambia `mars` por `iss`, y abajo, para `->visit()`: `/trip/iss`:

tests/Functional/BookingTest.php

```
// ... lines 1 - 12
class BookingTest extends KernelTestCase
{
// ... lines 15 - 19
    public function testCreateBooking(): void
    {
        $trip = TripFactory::createOne([
// ... line 23
            'slug' => 'iss',
// ... line 25
        ]);
// ... lines 27 - 30
        $this->browser()
// ... line 32
            ->visit('/trip/iss')
// ... lines 34 - 42
        ;
// ... lines 44 - 52
    }
}
```

¡Ejecuta de nuevo las pruebas!

```
bin/phpunit
```

¡Pasa!

Parece que nuestro correo se envía... ¡pero confirmémoslo! Al final de esta prueba, quiero hacer algunas afirmaciones sobre el correo electrónico. Symfony lo permite, pero a mí me gusta utilizar una biblioteca que devuelva la diversión a las pruebas funcionales de correo electrónico.

`zenstruck/mailer-test`

En tu terminal, ejecuta:

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

Instalado y configurado... de nuevo en nuestra prueba, habilítalo añadiendo el rasgo `InteractsWithMailer`:

tests/Functional/BookingTest.php

```
// ... lines 1 - 13
class BookingTest extends KernelTestCase
{
    use ResetDatabase, Factories, HasBrowser, InteractsWithMailer;
// ... lines 17 - 54
}
```

Empieza de forma sencilla, al final de la prueba, escribe `$this->mailer()->assertSentEmailCount(1);`:

tests/Functional/BookingTest.php

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

Variables de entorno específicas de la prueba

Nota rápida: `.env.local` -donde ponemos nuestras credenciales Mailtrap reales- no se lee ni se utiliza en el entorno `test`: nuestras pruebas sólo cargan `.env` y este archivo `.env.test`. Y en `.env`, `MAILER_DSN` está configurado como `null://null`. ¡Estupendo! Queremos que nuestras pruebas sean rápidas, y que no envíen realmente correos electrónicos.

¡Vuelve a ejecutarlas!

```
bin/phpunit
```

`assertEmailSentTo()`

Pasa: ¡se envía 1 correo electrónico! Vuelve atrás y añade otra aserción: `->assertEmailSentTo()`. ¿Qué dirección de correo esperamos? La que rellenamos en el formulario: `bruce@wayne-enterprises.com`. Cópiala y pégala. El segundo argumento es el asunto: `Booking Confirmation for Visit Mars`:

tests/Functional/BookingTest.php

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

¡Ejecuta las pruebas!

```
bin/phpunit
```

¡Sigue pasando! Y fíjate que ahora tenemos 20 afirmaciones en lugar de 19.

`TestEmail`

¡Pero podemos ir más allá! En lugar de una cadena para el asunto de esta afirmación, utiliza un cierre con `TestEmail $email` como argumento:

tests/Functional/BookingTest.php

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

Dentro, ahora podemos hacer muchas más afirmaciones sobre este correo electrónico. Como ya no estamos comprobando el asunto, añade primero ésta: `$email->assertSubject('Booking Confirmation for Visit Mars')`:

tests/Functional/BookingTest.php

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

¡Y podemos encadenar más afirmaciones!

Escribe `->assert` para ver qué sugiere nuestro editor. Míralas todas... Fíjate en `assertTextContains` y `assertHtmlContains`. Puedes aseverar sobre cada una de ellas por separado, pero, como es una buena práctica que ambas contengan los detalles importantes, utiliza `assertContains()` para comprobar las dos a la vez. Comprueba `Visit Mars`:

tests/Functional/BookingTest.php

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

Es importante comprobar los enlaces, así que asegúrate de que está la URL de reserva: `->assertContains('/booking/'.`. Ahora, `BookingFactory::first()->getUid()`:

tests/Functional/BookingTest.php

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

esto busca la primera entidad `Booking` en la base de datos (que sabemos por lo anterior que sólo hay una), y obtiene su `uid`.

Incluso podemos comprobar el archivo adjunto: `->assertHasFile('Terms of Service.pdf')`:

tests/Functional/BookingTest.php

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

Puedes comprobar el tipo de contenido y el contenido del archivo mediante argumentos adicionales, pero por ahora me basta con comprobar que el archivo adjunto existe.

¡Vamos, pruebas, vamos!

```
bin/phpunit
```

Impresionante, ¡25 aserciones ahora!

`->dd()`

Una última cosa: si alguna vez tienes problemas para averiguar por qué no pasa una de estas aserciones de correo electrónico, encadena un `->dd()`:

tests/Functional/BookingTest.php

```
// ... lines 1 - 14
class BookingTest extends KernelTestCase
{
// ... lines 17 - 21
    public function testCreateBooking(): void
    {
// ... lines 24 - 55
        $this->mailer()
// ... line 57
            ->assertEmailSentTo('bruce@wayne-enterprises.com', function(TestEmail $email) {
                $email
// ... lines 60 - 63
                    ->dd()
                ;
            })
        ;
    }
}
```

y ejecuta tus pruebas. Cuando llegue a ese `dd()`, vuelca el correo electrónico para ayudarte a depurar. ¡No olvides eliminarlo cuando hayas terminado!

A continuación, quiero añadir un segundo correo electrónico a nuestra aplicación. Para evitar la duplicación y mantener la coherencia, crearemos un diseño de correo electrónico Twig que ambos compartan.

Chapter 15: Diseño Twig de correo electrónico

¡Hora de una nueva función! Quiero enviar un correo electrónico recordatorio a los clientes 1 semana antes de su viaje reservado. ¡T menos 1 semana para despegar gente!

Problema con el Trabajador CLI de Symfony

En primer lugar, tenemos un pequeño problema con nuestro Symfony CLI worker. Abre `.symfony.local.yaml`. Nuestro trabajador `messenger` está buscando cambios en el directorio `vendor`. Al menos en algunos sistemas, hay demasiados archivos aquí para monitorizar y ocurren cosas raras. No pasa nada: elimina `vendor/`:

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

Y como hemos cambiado la configuración, salta a tu terminal y reinicia el servidor web:

```
symfony server:stop
```

Y

```
symfony serve -d
```

Diseño del correo electrónico

Nuestro nuevo correo electrónico de recordatorio de reserva tendrá una plantilla muy similar a la de confirmación de reserva. Para reducir la duplicación, y mantener la coherencia de nuestros elegantes correos electrónicos, en `templates/email/`, crea una nueva plantilla `layout.html.twig` a la que se extenderán todos nuestros correos electrónicos.

Copia el contenido de `booking_confirmation.html.twig` y pégalo aquí. Ahora, elimina el contenido específico de confirmación de reserva y crea un bloque `content` vacío. Creo que está bien mantener nuestra firma aquí.

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

En `booking_confirmation.html.twig`, aquí arriba, amplía este nuevo diseño y añade el bloque `content`. Abajo, copia el contenido específico del correo electrónico y pégalo dentro de ese bloque. Elimina todo lo demás.

templates/email/booking_confirmation.html.twig

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

{% block content %}
    <row>
        <columns>
            <spacer size="40"></spacer>
            <p class="accent-title">Get Ready for your trip to</p>
            <h1 class="trip-name">{{ trip.name }}</h1>
            <img
                    class="trip-image float-center"
                    src="{{ email.image('@images/%s.png'|format(trip.slug)) }}"
                    alt="{{ trip.name }}">
        </columns>
    </row>
    <row>
        <columns>
            <p class="accent-title">Departure: {{ booking.date|date('Y-m-d') }}</p>
        </columns>
    </row>
    <row>
        <columns>
            <button class="expanded rounded center" href="{{ url('booking_show', {uid: booking.uid}) }}">
                Manage Booking
            </button>
            <button class="expanded rounded center secondary" href="{{ url('bookings', {uid: customer.uid}) }}">
                My Account
            </button>
        </columns>
    </row>
{% endblock %}
```

Asegurémonos de que el correo electrónico de confirmación de la reserva sigue funcionando, ¡y tenemos pruebas para ello! De vuelta en el terminal, ejecútalas con:

```
bin/phpunit
```

¡Verde! Eso es buena señal. Asegurémonos doblemente comprobándolo en Mailtrap. En la aplicación, reserva un viaje... y comprueba Mailtrap. ¡Sigue estando fantástico!

¡Es hora de enviar el correo electrónico recordatorio!

Indicador de recordatorio de reserva

Después de enviar un correo electrónico recordatorio, tenemos que marcar la reserva para no molestar al cliente con múltiples recordatorios. Vamos a añadir una nueva bandera para esto a la entidad `Booking`.

En tu terminal, ejecuta:

```
symfony make:entity Booking
```

¡Uy!

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

¿Añadir un nuevo campo llamado `reminderSentAt`, tipo `datetime_immutable`, anulable? Sí. Se trata de un patrón habitual que utilizo para este tipo de campos bandera en lugar de un simple `boolean`. `null` significa `false` y una fecha significa `true`. Funciona igual, pero nos da un poco más de información.

Pulsa intro para salir del comando.

En la entidad `Booking`... aquí está nuestra nueva propiedad, y aquí abajo, el getter y el setter.

Encontrar reservas para recordar

A continuación, necesitamos una forma de encontrar todas las reservas que necesitan que se les envíe un recordatorio. ¡El trabajo perfecto para `BookingRepository`! Añade un nuevo método llamado `findBookingsToRemind()`, tipo de retorno: `array`. Añade un docblock para mostrar que devuelve un array de objetos Reserva:

src/Repository/BookingRepository.php

```
// ... lines 1 - 12
class BookingRepository extends ServiceEntityRepository
{
// ... lines 15 - 51
    /**
     * @return Booking[]
     */
    public function findBookingsToRemind(): array
    {
// ... lines 57 - 65
    }
}
```

Dentro, `return $this->createQueryBuilder()`, alias `b`. Encadena `->andWhere('b.reminderSentAt IS NULL')`, `->andWhere('b.date <= :future')`, `->andWhere('b.date > :now')` rellenando los marcadores de posición con `->setParameter('future', new \DateTimeImmutable('+7 days'))` y `->setParameter('now', new \DateTimeImmutable('now'))`. Termina con `->getQuery()->getResult()`:

src/Repository/BookingRepository.php

```
// ... lines 1 - 12
class BookingRepository extends ServiceEntityRepository
{
// ... lines 15 - 54
    public function findBookingsToRemind(): array
    {
        return $this->createQueryBuilder('b')
            ->andWhere('b.reminderSentAt IS NULL')
            ->andWhere('b.date <= :future')
            ->andWhere('b.date > :now')
            ->setParameter('future', new \DateTimeImmutable('+7 days'))
            ->setParameter('now', new \DateTimeImmutable('now'))
            ->getQuery()
            ->getResult()
        ;
    }
}
```

Fijación de Reservas Pendientes de Recordatorio

En `AppFixtures`, aquí abajo, creamos algunas reservas falsas. Añade una que desencadene con seguridad el envío de un correo electrónico recordatorio: `BookingFactory::createOne()`, dentro, `'trip' => $arrakis, 'customer' => $clark` y, ésta es la parte importante, `'date' => new \DateTimeImmutable('+6 days')`:

src/DataFixtures/AppFixtures.php

```
// ... lines 1 - 10
class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager): void
    {
// ... lines 15 - 87
        BookingFactory::createOne([
            'trip' => $arrakis,
            'customer' => $clark,
            'date' => new \DateTimeImmutable('+6 days'),
        ]);
    }
}
```

Claramente entre ahora y dentro de 7 días.

"Migración

Hemos realizado cambios en la estructura de nuestra base de datos. Normalmente, deberíamos crear una migración... pero, no estamos utilizando migraciones. Así que, simplemente forzaremos la actualización del esquema. En tu terminal, ejecuta:

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

Luego, vuelve a cargar los accesorios:

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

Todo ha funcionado, ¡genial!

A continuación, ¡crearemos un nuevo correo electrónico recordatorio y un comando CLI para enviarlo!

Chapter 16: Correo electrónico desde el comando CLI

Ya hemos hecho el trabajo previo para nuestra función de correo electrónico recordatorio. Ahora, ¡vamos a crear y enviar los correos!

Plantilla de correo electrónico recordatorio

En `templates/email`, la nueva plantilla de correo electrónico será muy similar a `booking_confirmation.html.twig`. Copia ese archivo y nómbralo `booking_reminder.html.twig`. Dentro, no quiero perder demasiado tiempo en esto, así que simplemente cambia el título del acento para que diga "¡Próximamente!":

templates/email/booking_reminder.html.twig

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

{% block content %}
    <row>
        <columns>
            <spacer size="40"></spacer>
            <p class="accent-title">Coming soon!</p>
            <h1 class="trip-name">{{ trip.name }}</h1>
            <img
                    class="trip-image float-center"
                    src="{{ email.image('@images/%s.png'|format(trip.slug)) }}"
                    alt="{{ trip.name }}">
        </columns>
    </row>
    <row>
        <columns>
            <p class="accent-title">Departure: {{ booking.date|date('Y-m-d') }}</p>
        </columns>
    </row>
    <row>
        <columns>
            <button class="expanded rounded center" href="{{ url('booking_show', {uid: booking.uid}) }}">
                Manage Booking
            </button>
            <button class="expanded rounded center secondary" href="{{ url('bookings', {uid: customer.uid}) }}">
                My Account
            </button>
        </columns>
    </row>
{% endblock %}
```

¡Envíalo! ¡Juego de palabras espacial accidental!

Comando Enviar Recordatorio

La lógica para enviar los correos electrónicos tiene que ser algo que podamos programar para que se ejecute cada hora o cada día. ¡El trabajo perfecto para un comando CLI! En tu terminal, ejecuta:

```
symfony make:command
```

¡Bah!

```
symfony console make:command
```

Llámalo: `app:send-booking-reminders`.

¡Ve a comprobarlo! `src/Command/SendBookingRemindersCommand.php`. Cambia la descripción a "Enviar correos electrónicos de recordatorio de reserva":

src/Command/SendBookingRemindersCommand.php

```
// ... lines 1 - 17
#[AsCommand(
// ... line 19
    description: 'Send booking reminder emails',
)]
class SendBookingRemindersCommand extends Command
// ... lines 23 - 70
```

En el constructor, autocablea y establece propiedades para `BookingRepository`, `EntityManagerInterface` y `MailerInterface`:

src/Command/SendBookingRemindersCommand.php

```
// ... lines 1 - 21
class SendBookingRemindersCommand extends Command
{
    public function __construct(
        private BookingRepository $bookingRepo,
        private EntityManagerInterface $em,
        private MailerInterface $mailer,
    ) {
        parent::__construct();
    }
// ... lines 31 - 68
}
```

Este comando no necesita argumentos ni opciones, así que elimina por completo el método `configure()`.

Limpia las tripas de `execute()`. Empieza añadiendo un bonito: `$io->title('Sending booking reminders')`. Luego, coge las reservas que necesitan que se envíen recordatorios, con `$bookings = $this->bookingRepo->findBookingsToRemind()`.

Barra de progreso fácil

Para ser los mejores, mostremos una barra de progreso mientras recorremos las reservas. El objeto `$io` tiene un truco para esto. Escribe `foreach ($io->progressIterate($bookings) as $booking)`. Esto se encarga de toda la aburrida lógica de la barra de progreso Dentro, tenemos que crear un nuevo correo electrónico. En `TripController`, copia ese correo electrónico -incluyendo estas cabeceras- y pégalo aquí.

Pero tenemos que ajustarlo un poco: elimina el archivo adjunto. Y para el asunto: sustituye "Confirmación" por "Recordatorio". Arriba, añade algunas variables por comodidad: `$customer = $booking->getCustomer()` y `$trip = $booking->getTrip()`. Aquí abajo, mantén los mismos metadatos, pero cambia la etiqueta a `booking_reminder`. Esto nos ayudará a distinguir mejor estos correos en Mailtrap.

Ah, y por supuesto, cambia la plantilla a `booking_reminder.html.twig`.

Siguiendo con el bucle, envía el correo electrónico con `$this->mailer->send($email)` y marca la reserva como recordatorio enviado con `$booking->setReminderSentAt(new \DateTimeImmutable('now'))`.

¡Perfecto! Fuera del bucle, llama a `$this->em->flush()` para guardar los cambios en la base de datos. Por último, celébralo con `$io->success(sprintf('Sent %d booking reminders', count($bookings)))`.

src/Command/SendBookingRemindersCommand.php

```
// ... lines 1 - 21
class SendBookingRemindersCommand extends Command
{
// ... lines 24 - 31
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $io = new SymfonyStyle($input, $output);

        $io->title('Sending booking reminders');

        $bookings = $this->bookingRepo->findBookingsToRemind();

        foreach ($io->progressIterate($bookings) as $booking) {
            $trip = $booking->getTrip();
            $customer = $booking->getCustomer();

            $email = (new TemplatedEmail())
                ->to(new Address($customer->getEmail()))
                ->subject('Booking Reminder for '.$trip->getName())
                ->htmlTemplate('email/booking_reminder.html.twig')
                ->context([
                    'customer' => $customer,
                    'trip' => $trip,
                    'booking' => $booking,
                ])
            ;

            $email->getHeaders()->add(new TagHeader('booking_reminder'));
            $email->getHeaders()->add(new MetadataHeader('booking_uid', $booking->getUid()));
            $email->getHeaders()->add(new MetadataHeader('customer_uid', $customer->getUid()));

            $this->mailer->send($email);
            $booking->setReminderSentAt(new \DateTimeImmutable('now'));
        }

        $this->em->flush();

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

        return Command::SUCCESS;
    }
}
```

¡Hora de probar! Ve a tu terminal. Para asegurarte de que tenemos una reserva que necesita que se le envíe un recordatorio, recarga los accesorios con:

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

Ahora, ¡ejecuta nuestro nuevo comando!

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

Bien, ¡se ha enviado 1 recordatorio! Y el resultado impresionará a nuestros colegas! Antes de comprobar Mailtrap, vuelve a ejecutar el comando:

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

"Enviados 0 recordatorios de reserva". ¡Perfecto! Nuestra lógica para marcar las reservas como recordatorios enviados ¡funciona!

Ahora comprueba Mailtrap... ¡aquí está! Como era de esperar, se parece mucho a nuestro correo de confirmación, pero aquí dice "Próximamente": está utilizando la nueva plantilla.

`X-Tag` y `X-Metadata`

Cuando se utiliza "Prueba de Mailtrap", las etiquetas y metadatos de Mailer no se convierten en categorías y variables personalizadas de Mailtrap, como ocurre cuando se envían en producción. ¡Pero aún puedes asegurarte de que se envían! Haz clic en esta pestaña "Información técnica" y desplázate un poco hacia abajo. Cuando Mailer no sabe cómo convertir las etiquetas y los metadatos, los añade como estas cabeceras genéricas personalizadas: `X-Tag` y `X-Metadata`.

Efectivamente, `X-Tag` es `booking_reminder`. Genial, ¡eso es lo que esperamos también!

Vale, ¿nueva función? ¡Comprobado! ¿Pruebas para la nueva función? ¡Eso a continuación!

Chapter 17: Prueba del comando CLI

¡El capitán está harto de que la gente corra detrás del cohete porque llegan tarde! ¡Por eso hemos creado un comando para enviar correos electrónicos recordatorios! Problema resuelto! Ahora escribamos una prueba para asegurarnos de que sigue funcionando. "Nueva función, nueva prueba", ¡ese es mi lema!

Salta a tu terminal y ejecuta:

```
symfony console make:test
```

Teclea? `KernelTestCase`. ¿Nombre? `SendBookingRemindersCommandTest`.

`SendBookingRemindersCommandTest`

En nuestro IDE, la nueva clase se ha añadido a `tests/`. Ábrelo y mueve la clase a un nuevo espacio de nombres: `App\Tests\Functional\Command`, para mantener las cosas organizadas.

Perfecto. Primero, limpia las tripas y añade algunos rasgos de comportamiento: `use ResetDatabase, Factories, InteractsWithMailer`:

tests/Functional/Command/SendBookingRemindersCommandTest.php

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

Elimina dos pruebas: `public function testNoRemindersSent()` con `$this->markTestIncomplete()` y `public function testRemindersSent()`. Márcalo también como incompleto:

tests/Functional/Command/SendBookingRemindersCommandTest.php

```
// ... lines 1 - 9
class SendBookingRemindersCommandTest extends KernelTestCase
{
// ... lines 12 - 13
    public function testNoRemindersSent()
    {
        $this->markTestIncomplete();
    }

    public function testRemindersSent()
    {
        $this->markTestIncomplete();
    }
}
```

De vuelta al terminal, ejecuta las pruebas con:

```
bin/phpunit
```

Lista de pruebas TODO

Fíjate, nuestras dos pruebas originales pasan, los dos puntos, y estas íes son las nuevas pruebas incompletas. Me encanta esta pauta: escribe los stubs de prueba para una nueva función, y luego juega a eliminar los incompletos uno a uno hasta que desaparezcan todos. Entonces, ¡la

funcionalidad está terminada!

Symfony tiene algunas herramientas para probar comandos, pero me gusta usar un paquete que las envuelve en una experiencia más agradable. Instálalo con:

`zenstruck/console-test`

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

Para activar los ayudantes de este paquete, añade un nuevo rasgo de comportamiento a nuestra prueba: `InteractsWithConsole`:

tests/Functional/Command/SendBookingRemindersCommandTest.php

```
// ... lines 1 - 10
class SendBookingRemindersCommandTest extends KernelTestCase
{
    use ResetDatabase, Factories, InteractsWithMailer, InteractsWithConsole;
// ... lines 14 - 26
}
```

¡Estamos listos para derribar esos yoes!

`testNoRemindersSent()`

La primera prueba es fácil: queremos asegurarnos de que, cuando no hay reservas que recordar, el comando no envía ningún correo electrónico. Escribe `$this->executeConsoleCommand()` y sólo el nombre del comando: `app:send-booking-reminders`. Asegúrate de que el comando se ejecuta correctamente con `->assertSuccessful()` y `->assertOutputContains('Sent 0 booking reminders')`:

tests/Functional/Command/SendBookingRemindersCommandTest.php

```
// ... lines 1 - 10
class SendBookingRemindersCommandTest extends KernelTestCase
{
// ... lines 13 - 14
    public function testNoRemindersSent()
    {
        $this->executeConsoleCommand('app:send-booking-reminders')
            ->assertSuccessful()
            ->assertOutputContains('Sent 0 booking reminders')
        ;
    }
// ... lines 22 - 26
}
```

`testRemindersSent()`

Organiza

Pasamos a la siguiente prueba Ésta es más complicada: tenemos que crear una reserva que pueda recibir un recordatorio. Crea el arreglo de la reserva con `$booking = BookingFactory::createOne()`. Pasa un array con `'trip' => TripFactory::new()`, y dentro de éste, otro array con `'name' => 'Visit Mars'`, `'slug' => 'iss'` (para evitar el problema de la imagen). La reserva también necesita un cliente: `'customer' => CustomerFactory::new()`. Lo único que nos importa es el correo electrónico del cliente: `'email' => 'steve@minecraft.com'` por último, la fecha de la reserva: `'date' => new \DateTimeImmutable('+4 days')`:

tests/Functional/Command/SendBookingRemindersCommandTest.php

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

¡Uf! Tenemos una reserva en la base de datos que necesita que se le envíe un recordatorio. El paso de configuración, u ordenación, de esta prueba está hecho.

Pre-Aserción

Añade una preafirmación para asegurarte de que no se ha enviado un recordatorio a esta reserva: `$this->assertNull($booking->getReminderSentAt())`:

tests/Functional/Command/SendBookingRemindersCommandTest.php

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

Actuar

Ahora el paso actuar: `$this->executeConsoleCommand('app:send-booking-reminders')->assertSuccessful()->assertOutputContains('Sent 1 bookir` :

tests/Functional/Command/SendBookingRemindersCommandTest.php

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

Afirma

Pasamos a la fase de aserción para asegurarnos de que el correo electrónico se ha enviado. En `BookingTest`, copia la aserción del correo electrónico y pégala aquí. Haz algunos ajustes: el correo electrónico es `steve@minecraft.com`, el asunto es

`Booking Reminder for Visit Mars` y este correo no tiene ningún adjunto, así que elimina esa aserción por completo:

tests/Functional/Command/SendBookingRemindersCommandTest.php

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

Por último, escribe una aserción de que el comando actualizó la reserva en la base de datos. `$this->assertNotNull($booking->getReminderSentAt())`:

tests/Functional/Command/SendBookingRemindersCommandTest.php

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

¡El momento de la verdad! Ejecuta las pruebas:

```
bin/phpunit
```

¡Todo en verde!

Pruebas externas

Este tipo de pruebas externas me parecen muy divertidas y fáciles de escribir, porque no tienes que preocuparte demasiado de probar la lógica interna e imitan la forma en que un usuario interactúa con tu aplicación. No es casualidad que las afirmaciones se centren en lo que el usuario debería ver y en algunas comprobaciones de alto nivel posteriores a la interacción, como comprobar algo en la base de datos.

Ahora que tenemos pruebas para nuestras dos rutas de envío de correo electrónico, demos una vuelta de la victoria y refactoricemos con confianza para eliminar la duplicación.

Chapter 18: Servicio de fábrica de correos electrónicos

Nuestra aplicación envía dos correos electrónicos: en `SendBookingRemindersCommand`, y en `TripController::show()`. Aquí hay... mucha duplicación. ¡Me duele la vista! ¡Pero no te preocupes! Podemos reorganizar esto en un servicio de fábrica de correos electrónicos. Y como tenemos pruebas que cubren ambos correos, podemos refactorizar y estar seguros de que no hemos roto nada. No me canso de decirlo: ¡me encantan las pruebas!

`BookingEmailFactory`

Empieza creando una nueva clase: `BookingEmailFactory` en el espacio de nombres `App\Email`. Añade un constructor, copia el argumento `$termsPath` de `TripController::show()`, pégalo aquí y conviértelo en una propiedad privada:

src/Email/BookingEmailFactory.php

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

Ahora, crea dos métodos de fábrica: `public function createBookingConfirmation()`, que aceptarán `Booking $booking`, y devolverán `TemplatedEmail`. Luego, `public function createBookingReminder(Booking $booking)` también devolverá un `TemplatedEmail`:

src/Email/BookingEmailFactory.php

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

Crea un método para albergar esa maldita duplicación: `private function createEmail()`, con argumentos `Booking $booking` y `string $tag` que devuelve un `TemplatedEmail`:

src/Email/BookingEmailFactory.php

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

Salta a `TripController::show()`, copia todo el código de creación del correo electrónico y pégalo aquí. Arriba, necesitamos dos variables: `$customer = $booking->getCustomer()` y `$trip = $booking->getTrip()`. Elimina `attachFromPath()`, `subject()`, y `htmlTemplate()`.

En este `TagHeader`, utiliza la variable `$tag` pasada. Podemos dejar los metadatos igual. Por último, devuelve el `$email`:

src/Email/BookingEmailFactory.php

```
// ... lines 1 - 11
class BookingEmailFactory
{
// ... lines 14 - 35
    private function createEmail(Booking $booking, string $tag): TemplatedEmail
    {
        $customer = $booking->getCustomer();
        $trip = $booking->getTrip();
        $email = (new TemplatedEmail())
            ->to(new Address($customer->getEmail()))
            ->context([
                'customer' => $customer,
                'trip' => $trip,
                'booking' => $booking,
            ])
        ;

        $email->getHeaders()->add(new TagHeader($tag));
        $email->getHeaders()->add(new MetadataHeader('booking_uid', $booking->getUid()));
        $email->getHeaders()->add(new MetadataHeader('customer_uid', $customer->getUid()));

        return $email;
    }
}
```

Con nuestra lógica compartida en su sitio, úsala en `createBookingConfirmation()`. Escribe `return $this->createEmail()`, pasando la variable `$booking` y `booking` para la etiqueta. Ahora, `->subject()`, copia esto de `TripController::show()`, cambiando la variable `$trip` por `$booking->getTrip()`. Por último, `->htmlTemplate('email/booking_confirmation.html.twig')`:

src/Email/BookingEmailFactory.php

```
// ... lines 1 - 11
class BookingEmailFactory
{
// ... lines 14 - 19
    public function createBookingConfirmation(Booking $booking): TemplatedEmail
    {
        return $this->createEmail($booking, 'booking')
            ->subject('Booking Confirmation for '.$booking->getTrip()->getName())
            ->htmlTemplate('email/booking_confirmation.html.twig')
        ;
    }
// ... lines 27 - 54
}
```

Para `createBookingReminder()`, copia el interior de `createBookingConfirmation()` y pégalo aquí. Cambia la etiqueta a `booking_reminder`, el asunto a `Booking Reminder`, y la plantilla a `email/booking_reminder.html.twig`:

src/Email/BookingEmailFactory.php

```
// ... lines 1 - 11
class BookingEmailFactory
{
// ... lines 14 - 19
    public function createBookingConfirmation(Booking $booking): TemplatedEmail
    {
        return $this->createEmail($booking, 'booking')
            ->subject('Booking Confirmation for '.$booking->getTrip()->getName())
            ->htmlTemplate('email/booking_confirmation.html.twig')
        ;
    }
// ... lines 27 - 54
}
```

El refactorizador

¡Ahora viene lo divertido! ¡Usar nuestra fábrica y eliminar un montón de código!

En `TripController::show()`, en lugar de inyectar `$termsPath`, inyecta `BookingEmailFactory $emailFactory`:

src/Controller/TripController.php

```
↕ // ... lines 1 - 18
19 final class TripController extends AbstractController
20 {
↕ // ... lines 21 - 29
30     public function show(
↕ // ... lines 31 - 35
36         BookingEmailFactory $emailFactory,
37     ): Response {
↕ // ... lines 38 - 58
59     }
60 }
```

Elimina todo el código de creación de correo electrónico y dentro de `$mailer->send()`, escribe `$emailFactory->createBookingConfirmation($booking)`:

src/Controller/TripController.php

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

En `SendBookingRemindersCommand`, de nuevo, elimina todo el código de creación de correo electrónico. Arriba en el constructor, autoconecta `private BookingEmailFactory $emailFactory`:

src/Command/SendBookingRemindersCommand.php

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

Aquí abajo, dentro de `$this->mailer->send()`, escribe `$this->emailFactory->createBookingReminder($booking)`:

src/Command/SendBookingRemindersCommand.php

```
// ... lines 1 - 18
class SendBookingRemindersCommand extends Command
{
// ... lines 21 - 29
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
// ... lines 32 - 37
        foreach ($io->progressIterate($bookings) as $booking) {
            $this->mailer->send($this->emailFactory->createBookingReminder($booking));
// ... line 40
        }
// ... lines 42 - 47
    }
}
```

Pruébalo

Oh, sí, ¡qué bien me ha sentado! ¿Pero hemos roto algo? Los canadienses tenemos fama de ser un poco salvajes. Compruébalo ejecutando las pruebas:

```
bin/phpunit
```

¡Uh oh, un fallo! Menos mal que tenemos estas pruebas, ¿eh?

El fallo viene de `BookingTest`:

> *"El mensaje no incluye un archivo con nombre de archivo [Condiciones del servicio.pdf]."*

Arréglalo

¡Fácil de arreglar! Durante nuestra refactorización, olvidé adjuntar el emocionante PDF de las condiciones del servicio al correo electrónico de confirmación de la reserva. Y nuestros clientes dependen de ello. Busca `BookingEmailFactory::createBookingConfirmation()`, y añade `->attachFromPath($this->termsPath, 'Terms of Service.pdf')`:

src/Email/BookingEmailFactory.php

```
// ... lines 1 - 11
class BookingEmailFactory
{
// ... lines 14 - 19
    public function createBookingConfirmation(Booking $booking): TemplatedEmail
    {
        return $this->createEmail($booking, 'booking')
// ... lines 23 - 24
            ->attachFromPath($this->termsPath, 'Terms of Service.pdf')
        ;
    }
// ... lines 28 - 55
}
```

Vuelve a ejecutar las pruebas:

```
bin/phpunit
```

¡Pasadas! ¿Reforzamiento satisfactorio? ¡Comprobado!

A continuación, vamos a cambiar un poco de marcha y sumergirnos en dos nuevos componentes Symfony para consumir los eventos webhook de correo electrónico de Mailtrap.

Chapter 19: El Componente Webhook para Eventos de Email

En Mailtrap, cuando enviamos correos electrónicos en producción, recordemos que podemos comprobar cada correo: si fue enviado, entregado, abierto, rebotado (¡lo cual es importante!) y más. Mailtrap nos permite establecer una URL de webhook para que nos envíe información sobre estos eventos.

Componentes Webhook y RemoteEvent

Como bonus, ¡descubrimos dos nuevos componentes de Symfony! Busca tu terminal e instálalos:

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

El componente webhook nos proporciona una única ruta a la que enviar todos los webhooks. Analiza los datos que se nos envían -llamados carga útil-, los convierte en un objeto de evento remoto y los envía a un consumidor. Puedes pensar en los eventos remotos como algo similar a los eventos Symfony. En lugar de que tu aplicación envíe un evento, lo hace un servicio de terceros, de ahí lo de evento remoto. Y en lugar de oyentes de eventos, decimos que los eventos remotos tienen consumidores.

Ejecuta

```
git status
```

para ver qué ha añadido la receta: `config/routes/webhook.yaml`. Eso añade el controlador webhook. Comprueba la ruta con:

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

Comprueba la primera. La ruta es `/webhook/{type}`. Así que ahora tenemos que configurar algún tipo.

los webhooks de terceros -como los de Mailtrap o los de un procesador de pagos o un sistema de alertas de Supernova- pueden enviarnos cargas útiles muy diferentes, por lo que normalmente necesitamos crear nuestros propios analizadores y eventos remotos. Dado que los eventos de correo electrónico son bastante estándar, Symfony proporciona algunos eventos remotos out-of-the-box para ellos: `MailerDeliveryEvent` y `MailerEngagementEvent`. Algunos puentes de correo, incluido el puente Mailtrap que estamos utilizando, proporcionan analizadores para cada carga útil de webhook del servicio para crear estos objetos. Sólo tenemos que configurarlo.

Configuración del analizador sintáctico Mailtrap

En `config/packages/`, crea un archivo `webhook.yaml`. Añade `framework`,`webhook`, `routing`, `mailtrap` (este es el tipo utilizado en la URL), y luego `service`. Para averiguar el id de servicio del analizador Mailtrap, ve a la documentación de Symfony Webhook. Busca el id de servicio del analizador Mailtrap, cópialo... y pégalo aquí:

config/packages/webhook.yaml

```
framework:
    webhook:
        routing:
            mailtrap:
                service: mailer.webhook.request_parser.mailtrap
```

`EmailEventConsumer`

Ahora necesitamos un consumidor. Crea una nueva clase llamada `EmailEventConsumer` en el espacio de nombres `App\Webhook`. Esto necesita implementar `ConsumerInterface` desde `RemoteEvent` Añade el método `consume()` necesario. Para decirle a Symfony qué tipo de webhook queremos que consuma, añade el atributo `#[AsRemoteEventConsumer]` con `mailtrap`:

src/Webhook/EmailEventConsumer.php

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

Sobre `consume()`, añade un docblock para ayudar a nuestro IDE: `@param MailerDeliveryEvent|MailerEngagementEvent $event`:

src/Webhook/EmailEventConsumer.php

```
// ... lines 1 - 11
class EmailEventConsumer implements ConsumerInterface
{
    /**
     * @param MailerDeliveryEvent|MailerEngagementEvent $event
     */
    public function consume(RemoteEvent $event): void
    {
// ... line 19
    }
}
```

Estos son los eventos remotos de correo genéricos que proporciona Symfony. Dentro, escribe `$event->` para ver los métodos disponibles.

En una aplicación real, aquí sería donde harías algo con estos eventos como guardarlos en la base de datos o notificar a un administrador si un correo electrónico rebota. En realidad, si un correo electrónico rebota varias veces, puede que quieras actualizar algo para evitar que se vuelva a intentar, ya que esto puede perjudicar la fiabilidad de tu correo electrónico. Pero para nuestros propósitos, basta con `dump($event)`:

src/Webhook/EmailEventConsumer.php

```
// ... lines 1 - 11
class EmailEventConsumer implements ConsumerInterface
{
// ... lines 14 - 16
    public function consume(RemoteEvent $event): void
    {
        dump($event);
    }
}
```

Consumidores asíncronos

Una última cosa: el controlador webhook envía el evento remoto al consumidor a través de Symfony Messenger, dentro de una clase de mensaje llamada `ConsumeRemoteEventMessage`.

Para manejar esto de forma asíncrona y mantener rápidas las respuestas de tu webhook, en `config/packages/messenger.yaml`, bajo `routing`, añade `Symfony\Component\RemoteEvent\Messenger\ConsumeRemoteEventMessage` y envíalo a nuestro transporte `async`:

config/packages/messenger.yaml

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

¡Vale! Estamos listos para hacer una demostración de este webhook. ¡Eso a continuación!

Chapter 20: Demostración de nuestro webhook a través de un agujero de gusano

¡Es hora de probar el webhook Mailtrap!

En primer lugar, tenemos que volver a cambiar nuestro entorno de desarrollo para enviar en producción. En `.env.local`, cambia a tu Mailtrap de producción `MAILER_DSN` y en `config/services.yaml`, asegúrate de que el dominio `global_from_email`'s es el que configuraste con Mailtrap.

Crea un Webhook en Mailtrap

En Mailtrap, ve a "Configuración" > "Webhooks" y haz clic en "Crear nuevo Webhook". Lo primero que necesitamos es una URL de Webhook. Hmm, esto tiene que ser `/webhook/mailtrap` pero tiene que ser una URL absoluta. En producción, esto no sería un problema: sería tu dominio de producción. En desarrollo, es un poco más complicado. No podemos utilizar simplemente la URL que nos da el servidor CLI de Symfony...

ngrok

De alguna manera tenemos que exponer nuestro servidor Symfony local al público. Y existe una herramienta muy útil que hace exactamente esto: ngrok. Crea una cuenta gratuita, inicia sesión y sigue las instrucciones para configurar el cliente CLI ngrok.

En el terminal, reinicia el servidor web Symfony:

```
symfony server:stop
```

No se está ejecutando. Inícialo con:

```
symfony serve -d
```

Exponer el servidor local

Esta es la URL que necesitamos exponer, cópiala y ejecútala:

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

Pega la URL y pulsa intro. ¡Agujero de gusano abierto!

Esta URL de "Reenvío" de aspecto loco es la URL pública. Cópiala y pégala en tu navegador. Esta advertencia sólo te permite saber que estás atravesando un túnel. Haz clic en "Visitar sitio" para ver tu aplicación. ¡Genial!

URL del Webhook de Mailtrap

De vuelta en Mailtrap, pega esta URL y añade `/webhook/mailtrap` al final. En "Seleccionar flujo", elige "Transaccional". En "Seleccionar dominio", elige tu dominio Mailtrap configurado. Selecciona todos los eventos y luego "Guardar".

Vuelve al nuevo webhook y haz clic en "Ejecuta la prueba".

> *"La prueba de la URL del webhook se ha completado correctamente"*

¡Buena señal!

Volcar el servidor

¿Recuerdas que en nuestro `EmailEventConsumer`, sólo volcamos el evento? Como el acceso al webhook se produce entre bastidores, no podemos ver el volcado... ¿o sí? Ejecuta en un nuevo terminal:

```
symfony console server:dump
```

Esto se conecta a nuestra aplicación y cualquier volcado se mostrará aquí en directo. ¡Inteligente!

En tu navegador, reserva un viaje, recuerda utilizar una dirección de correo electrónico real (¡pero no la mía!)

`MailerDeliveryEvent`

¡Momento de la verdad! De nuevo en el terminal ejecutando el servidor de volcado, espera un poco... ¡Muy bien! ¡Tenemos un volcado! Desplázate un poco hacia arriba... Se trata de un `MailerDeliveryEvent` para `delivered`. Vemos el ID interno que Mailtrap le asignó, la carga útil sin procesar, la fecha, el correo electrónico del destinatario, incluso nuestros metadatos y etiqueta personalizados.

`MailerEngagementEvent`

¡Probemos con un evento de compromiso! En tu cliente de correo electrónico, abre el correo.

De vuelta en el terminal del servidor de volcado, espera un poco... ¡y boom! ¡Otro evento! Esta vez, es un `MailerEngagementEvent` para `open`. ¡Qué guay!

Muy bien, cadetes espaciales, ¡esto es todo por este curso! Hemos conseguido cubrir casi todas las funcionalidades de Symfony Mailer sin SPAMear a nuestros usuarios. ¡Ganamos!

hasta la próxima, ¡feliz programación!

Chapter 21: Bonificación: Programar nuestro comando de correo electrónico

¿Todavía estás aquí? ¡Estupendo! Tengo un capítulo extra para ti.

Uno de nuestros becarios, Hugo, se queja de que tiene que conectarse a nuestro servidor y ejecutar el comando de recordatorio de reservas, todas las noches a medianoche. No sé cuál es el problema, ¿para eso no están los becarios?

Instalando el Programador de Symfony

Pero... Supongo que para ser más robustos, deberíamos automatizar esto por si está enfermo o se le olvida. Podríamos configurar una tarea CRON... pero eso no sería ni de lejos tan genial o flexible como usar el componente Programador de Symfony. Es perfecto para esto. En tu terminal, ejecuta:

```
composer require scheduler
```

Piensa en Symfony Scheduler como un complemento para Messenger. Proporciona su propio transporte especial que, en lugar de una cola, determina si es el momento de ejecutar un trabajo. Cada trabajo, o tarea, es un mensaje de Messenger, por lo que requiere un gestor de mensajes. Consumes la programación, como cualquier transporte de Messenger, con el comando `messenger:consume`.

`make:schedule`

Crea un horario con:

```
symfony console make:schedule
```

> **Note**
>
> `symfony/scheduler` ahora tiene una receta oficial que crea `src/Schedule.php` por ti, por lo que este paso ya no es necesario.

¿Nombre del transporte? Utiliza `default`. ¿Nombre del programa? Utiliza el predeterminado: `MainSchedule`. ¡Emocionante!

Es posible tener varios horarios, pero para la mayoría de las aplicaciones, un solo horario es suficiente.

Configurar el horario

Compruébalo: `src/Scheduler/MainSchedule.php`. Es un servicio que implementa `ScheduleProviderInterface` y está marcado con el atributo `#[AsSchedule]` con el nombre `default`. El creador inyectó automáticamente la caché, y veremos por qué en un segundo. El método `getSchedule()` es donde configuramos la programación y añadimos tareas.

Este `->stateful()` al que pasamos `$this->cache` es importante. Si el proceso que está ejecutando este programa se cae -como si nuestros trabajadores de Messenger se detuvieran temporalmente durante un reinicio del servidor-, cuando vuelva a estar en línea, sabrá todas las tareas que se ha saltado y las ejecutará. Si se suponía que una tarea debía ejecutarse 10 veces mientras estaba inactiva, las ejecutará todas. Esto puede no ser lo deseado, así que añade `->processOnlyLastMissedRun(true)` para que sólo se ejecute la última:

src/Scheduler/MainSchedule.php

```
// ... lines 1 - 12
final class MainSchedule implements ScheduleProviderInterface
{
// ... lines 15 - 19
    public function getSchedule(): Schedule
    {
        return (new Schedule())
// ... lines 23 - 29
            ->processOnlyLastMissedRun(true)
        ;
    }
}
```

¡A prueba de balas!

Para aplicaciones más complejas, puedes estar consumiendo el mismo programa en varios trabajadores. Utiliza `->lock()` para configurar un bloqueo de modo que sólo un trabajador ejecute la tarea cuando le corresponda.

Añadir una tarea

¡Es hora de añadir nuestra primera tarea! En `->add()`, escribe `RecurringMessage::`. Hay varias formas de activar una tarea. A mí me gusta utilizar `cron()`. Quiero que esta tarea se ejecute a medianoche, todos los días, así que utiliza `0 0 * * *`. El segundo argumento es el mensaje de Messenger a enviar. Queremos ejecutar `SendBookingRemindersCommand`, pero no podemos añadirlo aquí directamente. En su lugar, utiliza `new RunCommandMessage()` y pasa el nombre del comando: `app:send-booking-reminders` (aquí también puedes pasar argumentos y opciones):

src/Scheduler/MainSchedule.php

```
// ... lines 1 - 12
final class MainSchedule implements ScheduleProviderInterface
{
// ... lines 15 - 19
    public function getSchedule(): Schedule
    {
        return (new Schedule())
            ->add(
                RecurringMessage::cron(
                    '0 0 * * *',
                    new RunCommandMessage('app:send-booking-reminders')
                )
            )
// ... lines 29 - 30
        ;
    }
}
```

Depurar el programa

En tu terminal, lista las tareas de nuestro programa ejecutando:

```
symfony console debug:schedule
```

Tenemos un error.

> *"No puedes utilizar "CronExpressionTrigger" porque el paquete "cron expression" no está instalado"*

Solución fácil: copia el comando de instalación y ejecútalo:

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

¡Buen nombre! Ahora vuelve a ejecutar el comando de depuración:

```
symfony console debug:schedule
```

Aquí vamos, la salida está un poco torcida en esta pequeña pantalla, pero puedes ver la expresión cron, el mensaje (y el comando), y el próximo tiempo de ejecución: esta noche a medianoche.

`#[AsCronTask]`

Hay una alternativa para programar comandos. En `MainSchedule::getSchedule()`, borra el atributo `->add()`. Luego salta a nuestro `SendBookingRemindersCommand` y añade otro atributo: `#[AsCronTask()]` pasando a: `0 0 * * *`:

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

En tu terminal, depura de nuevo el horario para asegurarte de que sigue apareciendo:

```
symfony console debug:schedule
```

Y lo está, bastante bien.

Si tienes muchas tareas programadas a la misma hora, como a medianoche, puede que veas un pico de CPU a esta hora en tu servidor. A menos que sea superimportante que las tareas se ejecuten a una hora muy concreta, deberías repartirlas. Una forma de hacerlo, por supuesto, es asegurarte manualmente de que todas tienen expresiones cron diferentes, pero... eso es un rollo.

Expresiones de cron con hash

Para nuestro comando `app:send-booking-reminders`, no me importa cuándo se ejecuta, sólo que se ejecute una vez al día. Podemos utilizar una expresión cron con hash. En nuestra expresión, sustituye los 0 por #. El # significa "elige un valor aleatorio válido para esta parte":

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

Vuelve a depurar la programación:

```
symfony console debug:schedule
```

Está programado para ejecutarse a las 5:11 h. Ejecuta de nuevo el comando:

```
symfony console debug:schedule
```

Siguen siendo las 5:11 h. Vale, no es realmente aleatorio, los valores se calculan de forma determinista basándose en los detalles del mensaje. En nuestro caso, la cadena `app:send-booking-reminders`. Un comando diferente con la misma expresión hash tendrá valores diferentes.

La documentación del Programador tiene todos los detalles al respecto. Incluso hay alias para hashes comunes. Por ejemplo, `#mignight` elegirá una hora entre medianoche y las 3 de la madrugada. Utilízalo para nuestra expresión:

src/Command/SendBookingRemindersCommand.php

```
// ... lines 1 - 19
#[AsCronTask('#midnight')]
class SendBookingRemindersCommand extends Command
// ... lines 22 - 52
```

y vuelve a depurar la programación:

```
symfony console debug:schedule
```

Uy, una errata, lo arreglo y lo vuelvo a ejecutar:

```
symfony console debug:schedule
```

Ahora está programado para ejecutarse todos los días a las 2:11 h. ¡Genial!

Ejecutar la programación

Ya hemos configurado nuestro programa, pero ¿cómo lo ejecutamos? Recuerda que las programaciones no son más que transportes de Messenger. El nombre del transporte es `scheduler_<schedule_name>`, en nuestro caso, `scheduler_default`. Ejecútalo con:

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

En tu servidor de producción, configúralo para que se ejecute en segundo plano como un trabajador normal de Messenger.

Muy bien, éste es un breve resumen del componente Programador. Consulta la documentación para obtener más información

¡Feliz programación!

Chapter 22: Bonificación: Messenger Monitor Bundle

Hola, ¿sigues aquí? ¡Estupendo! ¡Hagamos un último capítulo extra!

Cuando tienes un montón de mensajes y programaciones ejecutándose en segundo plano, puede ser difícil saber qué está pasando. ¿Se están ejecutando mis trabajadores? ¿Se está ejecutando mi programación? ¿Y hacia dónde se está ejecutando? ¿Y los fallos? Tenemos registros, pero... registros. En lugar de eso, vamos a explorar un bundle genial que nos proporciona una interfaz de usuario para saber qué está pasando con nuestro ejército de robots trabajadores

Instalación

En tu terminal, ejecuta:

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

Te pide que instales una receta, di que sí. Vuelve a nuestro IDE y mira lo que se ha añadido.

En primer lugar, se ha añadido un `src/Schedule.php`. Esto no está relacionado con este bundle. Desde el último capítulo, en el que añadimos el `Symfony Scheduler`, ahora tiene una receta oficial que añade una programación por defecto. Como ya tenemos uno, elimina este archivo.

`MessengerMonitorController`

Se ha añadido un nuevo controlador: `src/Controller/Admin/MessengerMonitorController.php`. Se trata de un stub para habilitar la interfaz de usuario del bundle. Extiende este `BaseMessengerMonitorController` del bundle y añade un prefijo de ruta de `/admin/messenger`. También añade este atributo `#[IsGranted('ROLE_ADMIN')]`. Esto es muy importante para tus aplicaciones reales. Sólo quieres que los administradores del sitio accedan a la IU, ya que muestra información sensible. No tenemos seguridad configurada en esta app, así que eliminaré esta línea:

src/Controller/Admin/MessengerMonitorController.php

```
// ... lines 1 - 7
#[Route('/admin/messenger')]
class MessengerMonitorController extends BaseMessengerMonitorController
{
}
```

`ProcessedMessage`

`src/Entity/ProcessedMessage.php` es una nueva entidad añadida por la receta. También es un stub que extiende esta clase `BaseProcessedMessage` y añade una columna ID. Se utiliza para hacer un seguimiento del historial de tus mensajes de Messenger. Por cada mensaje procesado, se persiste una nueva de estas entidades. Pero no te preocupes, esto se hace en tu proceso worker, por lo que no ralentizará el frontend de tu aplicación.

Como tenemos una nueva entidad, deberíamos añadir una migración, pero no tengo migraciones configuradas para este proyecto. Así que en tu terminal, ejecuta:

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

Instalar dependencias opcionales

Antes de comprobar la interfaz de usuario, el bundle tiene dos dependencias opcionales que quiero instalar. La primera:

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

Esto hace que las marcas de tiempo de la interfaz de usuario sean legibles, como "hace 4 minutos". Siguiente:

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

Como estamos utilizando expresiones cron para nuestras tareas programadas, este paquete las hace legibles. Así, en lugar de "11 2 * * * *", lo mostrará como "todos los días a las 2:11 AM". ¡Estupendo!

¡Ya estamos listos! Inicia el servidor con:

```
symfony serve -d
```

Panel de control

Salta al navegador y visita `/admin/messenger`. Éste es el panel de control de Messenger Monitor

Este primer widget muestra los trabajadores en ejecución y su estado. Podemos ver que tenemos 1 trabajador en ejecución para nuestro transporte `async`. Éste es el que hemos configurado para que se ejecute con nuestro servidor Symfony CLI.

A continuación, vemos nuestros transportes disponibles, cuántos mensajes están en cola y cuántos trabajadores los están ejecutando. Observa que nuestro transporte `scheduler_default` no se está ejecutando. Esto es de esperar, ya que no lo hemos configurado para que se ejecute localmente.

Debajo, tenemos una instantánea de las estadísticas de las últimas 24 horas.

A la derecha, veremos los últimos 15 mensajes procesados. Por supuesto, ahora está vacío.

Todos estos widgets se actualizan automáticamente cada 5 segundos.

Programar

¡Vamos a crear un historial! En la barra superior, haz clic en `Schedule` (observa que el icono está en rojo para indicar que la programación no se está ejecutando). Es una especie de "comando `debug:schedule` más avanzado". Vemos nuestra única tarea programada: `RunCommandMessage` para `app:send-booking-reminders`. Utiliza un `CronExpressionTrigger` para ejecutarse "todos los días a las 2:11 AM". hasta ahora se ha ejecutado 0, pero podemos ejecutarla manualmente haciendo clic en "Activar"... y seleccionando nuestro transporte `async`.

"Detalles"

Vuelve al panel de control. Se ejecutó correctamente, tardó 58 ms y consumió 31 MB de memoria. Haz clic en "Detalles" para ver aún más información "Tiempo en cola", "Tiempo para gestionar", marcas de tiempo... un montón de cosas buenas.

Estas etiquetas son muy útiles para filtrar mensajes. Puedes añadir tus propias etiquetas, pero algunas las añade el bundle: `manual` `schedule:default:<hash>` , porque ejecutamos "manualmente" una tarea programada, `schedule`, porque era una tarea programada, `schedule:default`, porque forma parte de nuestra programación por defecto. es el identificador único de esta tarea programada.

A la derecha está el "resultado" del "manejador" del mensaje - en este caso, `RunCommandMessageHandler`. Diferentes gestores tienen diferentes resultados (algunos no tienen ninguno). Para éste, el resultado es el código de salida del comando y la salida.

> *“Enviados 0 recordatorios de reserva”*

Vamos a ejecutar de nuevo esta tarea, pero esta vez, con una reserva que necesita que se le envíe un recordatorio. De vuelta a tu terminal, vuelve a cargar nuestras instalaciones:

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

Vuelve al navegador. El panel de control está vacío ahora, pero eso era de esperar: al recargar nuestros dispositivos también se ha borrado nuestro historial de mensajes. Haz clic en "Programar" y luego en "Activar" en nuestro transporte "asíncrono".

De vuelta en el panel de control, ahora tenemos 2 mensajes. `RunCommandMessage` de nuevo pero haz clic en sus "Detalles":

> *“Enviado 1 recordatorio de reserva”*

Ahora nuestro segundo mensaje: `SendEmailMessage`. Este fue enviado por el comando. Haz clic en sus "Detalles" para ver la información relacionada con el correo electrónico de sus resultados. Observa la etiqueta, `booking_reminder`. El bundle detectó automáticamente que estábamos enviando un correo electrónico con una etiqueta "Mailer", por lo que la añadió aquí.

Transporta

En el menú superior, puedes hacer clic en "Transportes" para ver más detalles sobre los mensajes pendientes de cada uno (si procede). El transporte `failed` muestra los mensajes fallidos y te da la opción de reintentarlos o eliminarlos, ¡directamente desde la interfaz de usuario!

Historial

"Historial" es donde podemos filtrar los mensajes: Periodo, limitar a un intervalo de fechas concreto. Transporte, limitar a un transporte específico. Estado, mostrar sólo éxitos o fracasos. Programación, incluir o excluir los mensajes activados por una programación. Tipo de mensaje, filtrar por clase de mensaje.

Estadísticas

"Estadísticas" muestra un resumen de estadísticas por clase de mensaje y puede limitarse a un intervalo de fechas específico.

Purgar el historial de mensajes

Como probablemente puedas imaginar, si tu aplicación ejecuta muchos mensajes, nuestra tabla de historial puede llegar a ser realmente grande. El bundle proporciona algunos comandos para purgar mensajes antiguos.

En la documentación del bundle, desplázate hasta "messenger:monitor:purge" y copia el comando. Necesitamos programar esto... ¿pero cómo? Con el Programador de Symfony, ¡por supuesto! Abre `src/Scheduler/MainSchedule.php` y añade una nueva tarea con `->add(RecurringMessage::cron())`. Utiliza `#midnight` para que se ejecute diariamente entre medianoche y las 3 de la madrugada. Añade `new RunCommandMessage()` y pega el comando. Añade la opción `--exclude-schedules`:

src/Scheduler/MainSchedule.php

```
// ... lines 1 - 12
final class MainSchedule implements ScheduleProviderInterface
{
// ... lines 15 - 19
    public function getSchedule(): Schedule
    {
        return (new Schedule())
// ... lines 23 - 24
            ->add(RecurringMessage::cron(
                    '#midnight',
                    new RunCommandMessage('messenger:monitor:purge --exclude-schedules'),
                )
            )
// ... lines 30 - 34
        ;
    }
}
```

Esto purgará los mensajes con más de 30 días de antigüedad, excepto los mensajes activados por una programación. Esto es importante porque tus tareas programadas pueden ejecutarse una vez al mes o incluso una vez al año. Esto te permite mantener un historial de ellas independientemente de su frecuencia.

Purgar el Historial de Programaciones

Sin embargo, debemos limpiarlos. Así que, volviendo a los documentos, copia un segundo comando: `messenger:monitor:schedule:purge`. Y en la programación, añádelo con `->add(RecurringMessage::cron('#midnight', new RunCommandMessage()))` y pégalo:

src/Scheduler/MainSchedule.php

```
// ... lines 1 - 12
final class MainSchedule implements ScheduleProviderInterface
{
// ... lines 15 - 19
    public function getSchedule(): Schedule
    {
        return (new Schedule())
// ... lines 23 - 29
            ->add(RecurringMessage::cron(
                    '#midnight',
                    new RunCommandMessage('messenger:monitor:schedule:purge'),
                )
            )
        ;
    }
}
```

Esto purgará el historial de mensajes programados omitidos por el comando anterior, pero conservará las 10 últimas ejecuciones de cada uno.

Asegurémonos de que estas tareas se han añadido a nuestra programación. De vuelta en el navegador, haz clic en "Programar" y aquí están: nuestras dos nuevas tareas.

Para la tarea que ejecutamos manualmente antes, podemos ver el resumen de la última ejecución, los detalles e incluso su historial.

Muy bien, amigos Esto es un rápido repaso a `zenstruck/messenger-monitor-bundle`. Echa un vistazo a los docs para obtener más información sobre todas sus funciones.

hasta la próxima, ¡feliz monitorización!