

### **Video 3.13: Cách gửi email bằng PHP - Symfony Mailer - Giao thức email - Cài đặt Mailhog - Hướng dẫn PHP 8 đầy đủ.**

Trong bài học này, chúng ta sẽ tìm hiểu cách gửi email, một chút về giao thức email, cài đặt hộp thư địa phương để kiểm tra email và nhiều điều khác nữa.

Vậy làm thế nào để gửi email bằng PHP? Có cách sử dụng chức năng tích hợp gọi là "mail," nhưng chúng ta sẽ không học cách sử dụng nó do có giới hạn về tính năng, tệp đính kèm và khả năng giao email. Một cách khác để gửi email là sử dụng gói công cụ của bên thứ ba, và trước đây, SwiftMailer là một lựa chọn tốt cho việc này, nhưng nó đã rút lui. Vì vậy, chúng ta sẽ tìm hiểu về Symfony Mailer trong bài học này.

Trước khi chúng ta cài đặt Symfony Mailer, chúng ta cần hiểu một chút về các giao thức email như SMTP, POP3 và IMAP. SMTP viết tắt của "Simple Mail Transfer Protocol" và được sử dụng để gửi email từ máy khách email như Outlook đến máy chủ email hoặc từ máy chủ email này đến máy chủ email khác. Khi bạn viết email và nhấn nút "Gửi," thường thì khi sử dụng SMTP, máy khách hoặc người gửi mở một kết nối TCP đến máy chủ SMTP và sử dụng một số lệnh để giao tiếp. Máy chủ email thực hiện một số công việc và sau đó gửi email đến máy chủ email của người nhận hoặc mục tiêu. Máy chủ email của người nhận sẽ lưu email cho đến khi người dùng hoặc người nhận mở hộp thư đến để kiểm tra email.

Bây giờ, SMTP thường được sử dụng cùng với các giao thức khác vì SMTP dùng để gửi hoặc chuyển email. IMAP hoặc POP3 là các giao thức được sử dụng để truy xuất hoặc nhận email. POP3 viết tắt của "Post Office Protocol Version 3" và IMAP viết tắt của "Internet Message Access Protocol". Sự khác biệt giữa chúng là IMAP cung cấp nhiều tính năng hơn so với POP3. Ví dụ, POP3 mặc định sẽ xóa email trên máy chủ sau khi nó đã nhận nó, trong khi IMAP lại không xóa email trên máy chủ và cho phép đồng bộ hóa qua nhiều máy khách. Email với IMAP sẽ được lưu trên máy chủ. Có cách để cấu hình POP3 để giữ bản sao của email trên máy chủ, nhưng IMAP vẫn cung cấp nhiều tính năng hơn.

Bây giờ, chúng ta sẽ cài đặt Symfony Mailer, chúng ta sẽ sao chép lệnh composer require này

A screenshot of a terminal window with a dark background. The window has standard macOS window controls (red, yellow, green buttons) in the top right corner. The command `$ composer require symfony/mailer` is entered in the terminal.

```
$ composer require symfony/mailer
```

và chạy nó trong terminal của chúng ta. Chúng ta sẽ mở mã nguồn và mở terminal, sau đó chạy lệnh composer require, nó sẽ tải về tất cả các gói cần thiết, bao gồm cả Symfony Mailer.

Chúng ta có một bộ điều khiển người dùng với hai tuyến đường hoặc hai phương thức,

```
class UserController
{
    #[Get('/users/create')]
    public function create(): View
    {
        return View::make('users/register');
    }

    #[Post('/users')]
    public function register()
    {
        $name = $_POST['name'];
        $email = $_POST['email'];
        $firstName = explode(' ', $name)[0];

        $text = <<<Body
```

trong đó một là để tạo người dùng, đó là tuyến đường đăng ký người dùng, chỉ đơn giản là hiển thị biểu mẫu đăng ký. Sau đó, chúng ta có một phương thức đăng ký, đó là một yêu cầu POST đến địa chỉ của người dùng, mục tiêu của nó là đăng ký người dùng. Tuy nhiên, chúng ta sẽ không thực sự đăng ký người dùng, chúng ta chỉ sẽ gửi một email chào mừng và giả định rằng người dùng đã được đăng ký.

Để gửi email bằng Symfony Mailer, chúng ta cần một số thứ.

```
Body;

$email = (new Email())
    ->from('support@example.com')
    ->to($email)
    ->subject('Welcome!')
    ->text($text);
```

Chúng ta cần đối tượng transport hoặc transporter, một dịch vụ mailer và đối tượng thư email thực tế. Hãy tạo đối tượng thư email trước. Chúng ta sẽ tạo đối tượng email và gán nó bằng new email từ gói mime của Symfony, đó là một phần mềm phụ thuộc của Symfony Mailer. Chúng ta có thể sử dụng lớp này để xây dựng một email bằng cách nối các cuộc gọi phương thức. Do đó,

chúng ta cần thiết lập địa chỉ email nguồn bằng phương thức `from`. Chúng ta có thể thiết lập địa chỉ email đích bằng phương thức `to`, trong đó địa chỉ email này được lấy từ biểu mẫu. Chúng ta sẽ thiết lập chủ đề bằng phương thức `subject` là "Chào mừng", và thiết lập nội dung văn bản bằng phương thức `text` là nội dung email đã tạo. Đây là một đối tượng email rất đơn giản.

Bây giờ, khi chúng ta đã có đối tượng email, chúng ta cần gửi nó đi bằng cách sử dụng dịch vụ `mailer`. Chúng ta có thể tạo đối tượng `mailer` bằng cách sử dụng lớp `mailer`.

```
$mailer = new Mailer();
```

Nếu chúng ta nhấp vào lớp `mailer`, chúng ta sẽ thấy nó chỉ có một phương thức duy nhất có tên là "send," nhận một thông điệp làm đối số. Vì vậy, chúng ta chỉ cần gọi phương thức "send" và truyền thông điệp vào đó.

```
public function send(RawMessage $message, Envelope $envelope = null): void
```

như một đối số, vì vậy chúng ta sẽ quay lại bộ điều khiển và đơn giản chỉ cần gọi `mailer send` email.

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

Như bạn có thể thấy, lớp `mailer` có một tham số bắt buộc trong hàm tạo mà chúng ta đang không truyền, đó là đối tượng `transport`, mà thực hiện một giao diện `transport`.

```
public function __construct(TransportInterface $transport, MessageBusInterface $bus = null,
```

`Transporter` có trách nhiệm gửi thông điệp email bằng cách sử dụng một giao thức, đây là lý do tại sao chúng ta đã nói về SMTP và các giao thức email khác. Nếu chúng ta kiểm tra vào giao diện này, chúng ta sẽ thấy nó chỉ có một phương thức duy nhất là "send."

```
interface TransportInterface
{
    /**
     * @throws TransportExceptionInterface
     */
    public function send(RawMessage $message, Envelope $envelope = null): ?SentMessage;

    public function __toString(): string;
}
```

Nếu chúng ta kiểm tra phương thức "send" trên lớp `mailer`, chúng ta thấy rằng nó chỉ gọi phương thức "send" trên đối tượng `transport`.

```

public function send(RawMessage $message, Envelope $envelope = null): void
{
    if (null === $this->bus) {
        $this->transport->send($message, $envelope);
    }

    return;
}

```

Lý do mà đây là một giao diện là vì có nhiều transport có thể được sử dụng để gửi email. Nếu chúng ta mở tài liệu và cuộn xuống phần thiết lập transport, chúng ta thấy có nhiều transport khác mà chúng ta có thể sử dụng, như các transport tự nhiên hoặc transport của bên thứ ba như Gmail, MailChimp, Postmark và Grid, v.v.

DSN protocol	Example	Description
smtp	smtp://user:password@smtp.example.com:25	Mailer uses an SMTP server to send emails
sendmail	sendmail://default	Mailer uses the local sendmail binary to send emails
native	native://default	Mailer uses the sendmail binary and options configured in the <code>sendmail_path</code> setting of <code>php.ini</code> . On Windows hosts, Mailer fallbacks to <code>smtp</code> and <code>smtp_port</code> <code>php.ini</code> settings when <code>sendmail_path</code> is not configured.

Chúng ta thấy có ba transport tích hợp sẵn: SMTP, Sendmail và Native. Trong bài học này, chúng ta sẽ sử dụng SMTP.

Vì vậy, chúng ta cần tạo một đối tượng transport và truyền nó vào hàm tạo của lớp mailer. Có một lớp transport mà chúng ta có thể sử dụng, nó có một phương thức tĩnh gọi là "fromDsn" có thể tạo một đối tượng transport cho chúng ta.

```

$transport = Transport::fromDsn();

```

Chúng ta cần truyền một chuỗi DSN (Data Source Name) và sau đó chúng ta sẽ sử dụng đối tượng transport này và truyền nó vào lớp mailer.

```
public static function fromDsn(string $dsn, EventDispatcherInterface $dispatcher = null,
```

Hãy kiểm tra lớp transport, bây giờ chúng ta cần một chuỗi DSN (Data Source Name) đúng đắn.

Để tạo đối tượng transport đúng đắn, DSN viết tắt của "Data Source Name," đó là một chuỗi biểu thị vị trí nơi có kết nối cơ sở dữ liệu, vị trí hệ thống tệp hoặc trong trường hợp này, thông tin truyền tải email chúng ta có thể thấy các DSN khác nhau trong tài liệu Symfony Mailer.

DSN protocol	Example	Description
smtp	<code>smtp://user:pass@smtp.example.com:25</code>	Mailer uses an SMTP server to send emails
sendmail	<code>sendmail://default</code>	Mailer uses the local sendmail binary to send emails
native	<code>native://default</code>	Mailer uses the sendmail binary and options configured in the <code>sendmail_path</code> setting of <code>php.ini</code> . On Windows hosts, Mailer fallbacks to <code>smtp</code> and <code>smtp_port</code> <code>php.ini</code> settings when <code>sendmail_path</code> is not configured.

Tất nhiên, chúng ta có thể sử dụng SMTP của Gmail để gửi email, nhưng thực tế chúng ta không muốn gửi email từ môi trường cục bộ của chúng ta. Chúng ta muốn kiểm tra email của mình xem chúng đã được gửi đi hay chưa, nhưng chúng ta không muốn gửi chúng đến hộp thư email thực tế. Vì vậy, chúng ta sẽ không sử dụng SMTP của Gmail, chúng ta cần một máy chủ SMTP cục bộ của riêng mình.

Vì vậy, chuỗi DSN của chúng ta sẽ giống như sau.

```
$dsn = 'smtp://user:pass@smtp.example.com:25';
```

Chúng ta sẽ sao chép nó và tạo một biến DSN và đặt nó vào. Bây giờ chúng ta cần một tên người dùng, mật khẩu và cổng cho máy chủ SMTP vì chuỗi trước đó không hợp lệ vì đây không phải là một máy chủ SMTP hợp lệ.

Như chúng ta đã đề cập trước đó, với máy chủ SMTP, chúng ta tất nhiên có thể sử dụng SMTP của Gmail của riêng mình, nhưng chúng ta sẽ không làm như vậy vì chúng ta không muốn gửi email thực sự từ môi trường cục bộ của mình. Chúng ta cần một hộp thư cục bộ nào đó để kiểm tra email, có một số dịch vụ miễn phí và trả phí cho điều đó, như Mailtrap, nhưng cũng có các gói mã nguồn mở như MailHog.



Chúng ta sẽ sử dụng MailHog và cài đặt nó thông qua Docker cho môi trường cục bộ của chúng ta, sau đó cấu hình nó. Tôi sẽ mở tệp docker-compose.yml và đã dán đoạn mã sau cho container MailHog.

```
mailhog:
  container_name: programwithgio-mailhog
  image: mailhog/mailhog
  restart: always
  logging:
    driver: "none"
  ports:
    - "8025:8025"
    - "1025:1025"
```

Chúng ta có tên container, hình ảnh là mailhog/mailhog và chúng ta đang tiết lộ hai cổng, 80 và 25. Cổng 80 dùng cho giao diện web mà chúng ta có thể truy cập để xem hộp thư đến của mình và cổng 1025 dùng cho máy chủ SMTP. Bây giờ chúng ta sẽ thoát khỏi container này, di chuyển vào thư mục docker và chạy lệnh `docker-compose up` để bắt đầu ứng dụng Docker.

```
root@e6de327345b8:/var/www# exit
exit

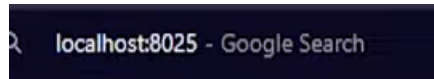
ggela@DESKTOP-45V9RT2 MINGW64 /d/code/learnphptherightway-project (3.11)
$ cd docker/

ggela@DESKTOP-45V9RT2 MINGW64 /d/code/learnphptherightway-project/docker (3.11)
$ docker-compose up -d --build
```

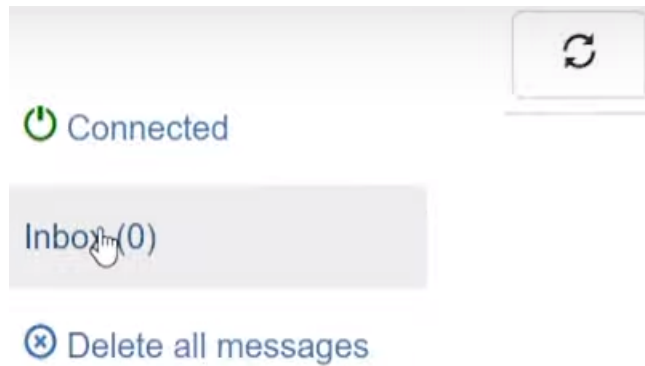
Và chúng ta đợi cho đến khi nó tải xong container MailHog. Trong lúc đó, khi MailHog đang cài đặt, chúng ta cần cập nhật chuỗi DSN SMTP. Thực ra, chúng ta không cần tên người dùng và mật khẩu, nhưng máy chủ SMTP của chúng ta là MailHog vì đó là container mà MailHog sẽ chạy ở trên. Vì vậy, chúng ta sẽ sử dụng "mailhog" cho máy chủ SMTP và cổng sẽ là 1025.

```
$dsn = 'smtp://mailhog:1025|';
```

Bây giờ chúng ta sẽ truyền biến DSN và chúng ta đã sẵn sàng. Hãy mở trình duyệt và truy cập localhost 8025.



Và chúng ta sẽ đi tới localhost cổng 8025 vì đó là nơi giao diện web chạy. Như bạn có thể thấy, nó đã mở MailHog và chúng ta có hộp thư đến của mình, nhưng hiện tại không có email nào trong đó.



Vậy thì chúng ta hãy thử nghiệm và gửi một số email.

Chúng ta nhấp vào nút "register," nó mất một giây và chúng ta nhận được một trang trống, điều đó có nghĩa rằng email đã được gửi và chúng ta không nhận được bất kỳ lỗi nào. Nếu chúng ta mở MailHog và làm mới hộp thư đến, chúng ta thấy email đã được nhận. Thêm vào việc gửi email dưới dạng văn bản

From support@example.com  
Subject **Welcome!**  
To gio@example.com

Plain text

Source

Hello gio,

Thank you for signing up!

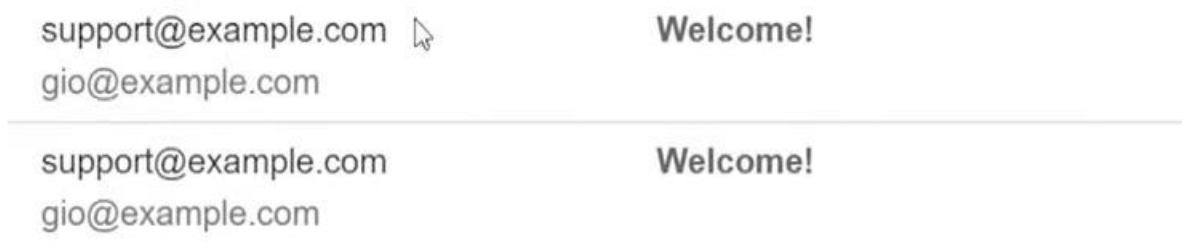
Chúng ta cũng có thể gửi HTML vì chúng ta không muốn chỉ gửi văn bản thuần túy như vậy. Chúng ta có thể muốn sử dụng các mẫu HTML và vân vân. Hãy tạo phiên bản HTML của đoạn văn bản này.

```
Body;  
  
$html = <<<HTMLBody  
<h1 style="text-align: center; color: blue;">Welcome</h1>  
Hello $firstName,  
<br />  
Thank you for signing up!  
HTMLBody;  
  
$email = (new Email())  
->from('support@example.com')  
->to($email)  
->subject('Welcome!')  
->text($text)  
->html($html);
```



Chúng ta sẽ sao chép cùng đoạn văn bản và có thể muốn thêm một thông báo chào mừng nào đó bằng màu xanh ở giữa nó và chúng ta cũng muốn thêm một dòng kẻ nghỉ. Sau đó, chúng ta sẽ nối thêm một phương thức khác ở đây gọi là "html" và truyền vào nội dung HTML.

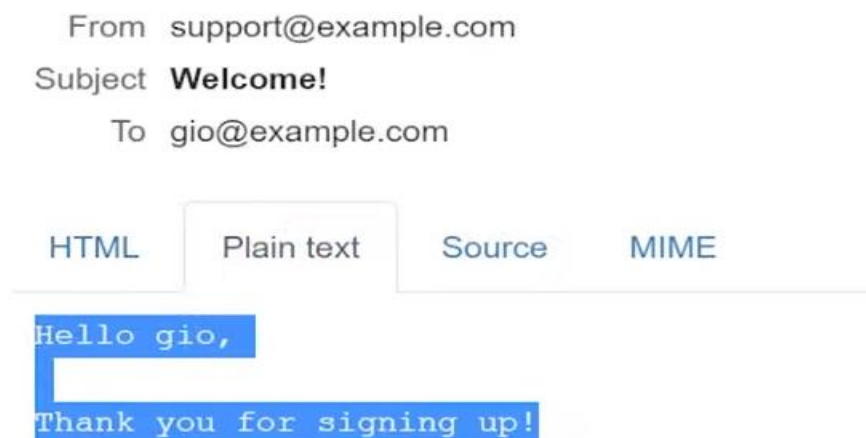
Hãy thử nghiệm điều này. Chúng ta sẽ gửi lại biểu mẫu cùng thông tin mà chúng ta đã gửi trước đó. Vậy là nó sẽ gửi email đến người dùng cùng thông tin. Nếu chúng ta làm mới hộp thư đến, chúng ta sẽ thấy email đã được gửi.



Và nếu chúng ta mở email, chúng ta sẽ thấy bây giờ chúng ta có một tab mới ở đây gọi là "html," và chúng ta nhận được tin nhắn email dưới dạng HTML.



Nếu chúng ta chuyển sang chế độ văn bản thuần túy, chúng ta vẫn có văn bản thuần túy sẵn sàng.




Lý do tại sao chúng ta đang sử dụng cả phiên bản HTML và phiên bản văn bản thuần túy trong mã nguồn là nếu một ứng dụng email không hỗ trợ HTML vì bất kỳ lý do nào, nó sẽ chuyển sang phiên bản văn bản thuần túy. Chúng ta cũng có thể gửi email với các tệp đính kèm, có thể đính kèm nội dung hoặc các tệp từ một đường dẫn cụ thể.

Chúng ta có thể nói thêm một phương thức gọi là "attach" chúng ta có thể đính kèm từ đường dẫn và chúng ta có thể đính kèm "part" hoặc chúng ta đơn giản chỉ đính kèm từ một nguồn hoặc nội dung. Trong trường hợp này, chúng ta chỉ đơn giản đính kèm một số nội dung, chúng ta sẽ nói "Hello world" và gọi nó là "welcome.txt".

```
$email = (new Email())
    ->from('support@example.com')
    ->to($email)
    ->subject('Welcome!')
    ->attach('Hello World!', 'welcome.txt')
    ->text($text)
    ->html($html);
```

Chúng ta sẽ làm mới trang web lại một lần nữa, nó sẽ gửi lại biểu mẫu cùng giá trị trước đó. Làm mới hộp thư đến và chúng ta đã nhận được email.

support@example.com gio@example.com	 <b>Welcome!</b>
support@example.com gio@example.com	<b>Welcome!</b>
support@example.com gio@example.com	<b>Welcome!</b>

Và nếu chúng ta chuyển sang MIME, chúng ta có tệp đính kèm ngay đây.

From support@example.com

Subject **Welcome!**


To gio@example.com

HTML


Plain text

Source


MIME

 Download

multipart/alternative; boundary=W5dFS0To (418 bytes)

 Download

application/octet-stream; name=welcome.txt (184 bytes)

 Download

Unknown type (2 bytes)

Chúng ta đang hardcode chuỗi DSN và đó không phải là cách đúng. Chúng ta muốn lấy nó từ biến môi trường. Vì vậy, chúng ta sẽ mở tệp .env và tạo biến môi trường "MAILER\_DSN" và đặt nó thành giá trị đó.

```
DB_HOST=db
DB_USER=root
DB_PASS=root
DB_DATABASE=programwithgio
MAILER_DSN=smtp://mailhog:1025|
```

Sau đó, chúng ta có thể tạo biến `MAILER\_DSN` từ biến siêu toàn cục `\$\_ENV`. Vì vậy, chúng ta sẽ thay thế dòng mã `\$dsn` bằng `\$dsn = \$\_ENV['MAILER\_DSN'];`

```
$transport = Transport::fromDsn($_ENV['MAILER_DSN']);
```

Và hãy đảm bảo rằng mọi thứ vẫn hoạt động. Chúng ta sẽ làm mới trang web và gửi lại biểu mẫu cùng giá trị trước đó, và chúng ta đã nhận được email.

Cải tiến tiếp theo mà chúng ta muốn thực hiện là đối tượng transport và lớp dịch vụ mailer có vẻ không phù hợp.

```
$transport = Transport::fromDsn($_ENV['MAILER_DSN']);
$mailer = new Mailer($transport);
```

Và sẽ loại bỏ nó. Chúng ta đã nói về dependency injection và các container dependency injection và điều này không theo đúng cách và thay vào đó làm điều gì đó như `mailer send`.

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

Và chấp nhận đối tượng mailer hoặc dịch vụ mailer thông qua constructor.

```
public function __construct(protected Mailer $mailer)
{
}
}
```

Bây giờ, nếu chúng ta kiểm tra lớp `mailer`, chúng ta thấy rằng nó triển khai giao diện `Mailer`.

```
final class Mailer implements MailerInterface
{
    private $transport;
    private $bus;
    private $dispatcher;

    public function __construct(TransportInterface $transport, MessageBusInterface $bus, DispatcherInterface $dispatcher)
    {
        $this->transport = $transport;
        $this->bus = $bus;
        $this->dispatcher = $dispatcher;
    }
}
```

Vì vậy, chúng ta không cần sử dụng một lớp cụ thể, chúng ta có thể đơn giản là mong đợi một loại triển khai của giao diện `Mailer`.

Tất nhiên, điều này không hoạt động vì chúng ta chưa nói cho ứng dụng của mình cách giải quyết giao diện `Mailer`. Chúng ta cần liên kết nó với một triển khai cụ thể. Vì vậy, nếu chúng ta mở lớp `app.php`.

```
class App
{
    private static DB $db;

    public function __construct(
        protected Container $container,
        protected Router $router,
        protected array $request,
        protected Config $config
    ) {
        static::$db = new DB($config->db ?? []);

        $this->container->set(PaymentGatewayServiceInterface::class, PaymentGatewayService::class);
    }
}
```

Ở đây, chúng ta thấy rằng chúng ta đang liên kết một giao diện dịch vụ cổng thanh toán với một lớp cụ thể dịch vụ cổng thanh toán. Vì vậy, chúng ta cần làm tương tự cho giao diện `Mailer` và chúng ta sẽ liên kết nó với một lớp `CustomMailer`.

```
class App
{
    private static DB $db;

    public function __construct(
        protected Container $container,
        protected Router $router,
        protected array $request,
        protected Config $config
    ) {
        static::$db = new DB($config->db ?? []);

        $this->container->set(PaymentGatewayServiceInterface::class, PaymentGatewayService::class);
        $this->container->set(MailerInterface::class, CustomMailer::class);
    }
}
```

Hãy tạo lớp này và chúng ta cần triển khai giao diện `Mailer`. Giao diện `Mailer` cung cấp một phương thức duy nhất là `send`, vì vậy chúng ta cần cung cấp triển khai cho phương thức đó.

```
use Symfony\Component\Mailer\Envelope;
use Symfony\Component\Mailer\MailerInterface;
use Symfony\Component\Mailer\Transport;
use Symfony\Component\Mailer\Transport\TransportInterface;
use Symfony\Component\Mime\RawMessage;

class CustomMailer implements MailerInterface
{
    protected TransportInterface $transport;

    public function __construct(protected string $dsn)
    {
        $this->transport = Transport::fromDsn($dsn);
    }

    public function send(RawMessage $message, Envelope $envelope = null): void
    {
        $this->transport->send($message, $envelope);
    }
}
```

Trong constructor, chúng ta có thể chấp nhận `dsn` và chúng ta sẽ gán nó thành một thuộc tính. Trong constructor, chúng ta có thể tạo đối tượng `transport`, vì vậy chúng ta sẽ dán mã chúng ta đã sao chép trước đó và đơn giản là tạo đối tượng `transport` bằng phương thức `fromDsn`. Thay vì sử dụng siêu biến môi trường (`\$\_ENV`), chúng ta sẽ truyền chuỗi `dsn` chúng ta nhận từ constructor. Chúng ta cũng có thể gán giá trị này thành một thuộc tính.

Trong phương thức `send`, chúng ta có thể gọi phương thức `send` trên đối tượng `transport` và truyền thông điệp vào. Vì vậy, chúng ta đang thực hiện cùng một điều với lớp `Mailer` khác.

Bây giờ chúng ta cũng cần chỉnh sửa lớp cấu hình của chúng ta. Vì nếu chúng ta quay lại `app.php`,

```
$this->container->set(MailerInterface::class, CustomMailer::class);
```

Đoạn mã trên sẽ không hoạt động vì lớp `Mailer` tùy chỉnh lấy một chuỗi `dsn` làm đối số. Thay vì làm như vậy, chúng ta có thể liên kết nó với một closure và tạo đối tượng `Mailer` tùy chỉnh và truyền chuỗi `dsn` từ lớp cấu hình. Vì vậy, chúng ta sẽ làm `config mailer dsn`.

```
$this->container->set(MailerInterface::class, fn() => new CustomMailer($config->mailer['dsn']));
```

Chúng ta cần thêm khả năng truy xuất cấu hình `mailer\_dsn` từ lớp cấu hình. Nếu chúng ta mở lớp cấu hình

```
public function __construct(array $env)
{
    $this->config = [
        'db' => [
            'host' => $env['DB_HOST'],
            'user' => $env['DB_USER'],
            'pass' => $env['DB_PASS'],
            'database' => $env['DB_DATABASE'],
            'driver' => $env['DB_DRIVER'] ?? 'mysql',
        ],
    ];
}
```

chúng ta thấy rằng chúng ta chỉ có tùy chọn `db`. Chúng ta chỉ cần tạo mục ở đây cho `mailer` và đặt nó thành một mảng. Sau đó, trong mảng đó, chúng ta sẽ đặt nó thành `dsn` và chúng ta sẽ lấy giá trị từ mảng biến môi trường mà chúng ta nhận được trong constructor. Vậy, chúng ta sẽ thực hiện `config mailer dsn`.

```
public function __construct(array $env)
{
    $this->config = [
        'db' => [
            'host' => $env['DB_HOST'],
            'user' => $env['DB_USER'],
            'pass' => $env['DB_PASS'],
            'database' => $env['DB_DATABASE'],
            'driver' => $env['DB_DRIVER'] ?? 'mysql',
        ],
        'mailer' => [
            'dsn' => $env['MAILER_DSN'] ?? '',
        ]
    ];
}
```

Và lý do tại sao nó đang được gạch chân ở đây

```
$this->container->set(PaymentGatewayServiceInterface::class, PaymentGatewayService::class);
$this->container->set(MailerInterface::class, fn() => new CustomMailer($config->mailer['dsn']));
}
```

là vì chúng ta chưa thêm thuộc tính đọc `phpdoc` vào đây.

```
/**
 * @property-read ?array $db
 * @property-read ?array $mailer
 */
```

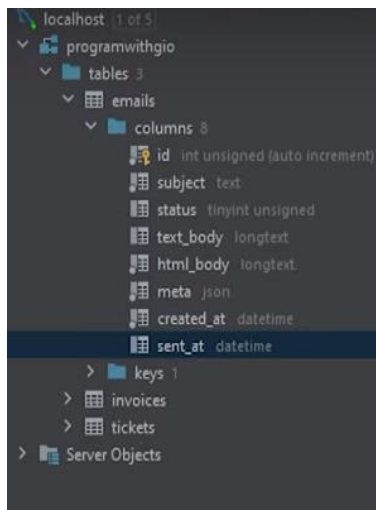
Hãy đóng nó lại và chúng ta hãy thử nghiệm để đảm bảo rằng mọi thứ vẫn hoạt động. Chúng ta sẽ mở trình duyệt, gửi lại biểu mẫu một lần nữa với cùng các giá trị. Chúng ta làm mới hộp thư đến và chúng ta nhận được email. Vậy là xong.



### Video 3.14: Cách lên lịch gửi email trong PHP - Chạy tập lệnh tự động bằng CRON - Hướng dẫn PHP 8 đầy đủ

Gửi email có thể là một quá trình nặng và có thể mất vài giây. Chúng ta không muốn người dùng phải chờ đợi khi chúng ta đang gửi email. Thay vào đó, chúng ta muốn xếp hàng email và có một loại kịch bản PHP để chạy trong nền tại một lịch trình cụ thể, sau đó có thể lấy email trong hàng đợi và gửi chúng đi.

Chúng ta tạo một bảng mới phía sau gọi là "emails," đơn giản lưu trữ chủ đề, trạng thái email, văn bản và nội dung HTML, dữ liệu siêu dữ liệu và một số bước thời gian như ngày tạo và ngày gửi. Chúng ta cũng đã tạo một mô hình email với một phương thức xếp hàng (queue method) để chèn email vào bảng. Nó chấp nhận đối tượng địa chỉ cho các biến "from" và "to," và đối tượng địa chỉ này đến từ gói Symfony, để làm việc vì chúng ta có thể đặt tên cùng với email và vấn đề liên quan.



Chúng ta cũng chấp nhận chủ đề và nội dung HTML và văn bản của email. Đương nhiên, trong ứng dụng thực tế, chúng ta có thể chấp nhận nhiều thông tin hơn, nhưng đối với ví dụ này, đó là đủ để không làm cho mọi thứ phức tạp hơn. Sau đó, chúng ta chỉ cần chuẩn bị truy vấn chèn và lưu "from" và "to" trong mảng metadata của chúng ta

```
use Symfony\Component\Mime\Address;

class Email extends Model
{
    public function queue(
        Address $to,
        Address $from,
        string $subject,
        string $html,
        ?string $text = null
    ): void {
        $stmt = $this->db->prepare(
            'INSERT INTO emails (subject, status, html_body, text_body, meta)
            VALUES (?, ?, ?, ?, ?, NOW())'
        );

        $meta['to'] = $to->toString();
        $meta['from'] = $from->toString();

        $stmt->execute([$subject, EmailStatus::Queue->value, $html, $text, $meta]);
    }
}
```

đây là một cột JSON trong bảng. Chúng ta cũng sử dụng "email status enum" để đặt trạng thái của email. Khi bản ghi email được tạo, nó sẽ được đặt thành "queue" theo mặc định.

Nếu chúng ta mở "email status enum," chúng ta thấy rằng đó là một enum với ba trạng thái: "queue," "sent," và "failed."

```
<?php
declare(strict_types = 1);

namespace App\Enums;

enum EmailStatus: int
{
    case Queue = 0;
    case Sent = 1;
    case Failed = 2;

    public function toString(): string
    {
        return match ($this) {
            self::Queue => 'In Queue',
            self::Sent => 'Sent',
            self::Failed => 'Failed'
        };
    }
}
```

Hãy mở lớp `UserController` và thay vì gửi email ngay lập tức, chúng ta có thể tạo một bản ghi trong cơ sở dữ liệu để gửi sau. Vì vậy, chúng ta sẽ lấy đoạn mã này

```
$email = (new Email())
    ->from('support@example.com')
    ->to($email)
    ->subject('Welcome!')
    ->attach('Hello World!', 'welcome.txt')
    ->text($text)
    ->html($html);
```

và thay vì thực hiện điều gì đó như `new Email(...)`

```
Body;

// ... $html = <<<HTMLBody
<h1 style="text-align: center; color: blue;">Welcome</h1>
Hello $firstName,
<br /><br />
Thank you for signing up!
HTMLBody;

// ... (new \App\Models\Email())->queue(
    new Address($email),
    new Address('support@example.com', 'Support'),
    'Welcome!',
    $html,
    $text
);
```

chúng ta sẽ tạo một `new EmailQueue(...)`. Chúng ta sẽ truyền vào các đối số cần thiết. Trước hết, chúng ta cần tạo một đối tượng địa chỉ mới, vì vậy chúng ta sẽ thực hiện `new Address(...)`, và chúng ta sẽ truyền vào địa chỉ email, trong trường hợp này là `email`. Hiện tại chúng ta không có tên, vì vậy chúng ta sẽ để trống phần đó. Sau đó, chúng ta cần địa chỉ email "from," vì vậy chúng ta sẽ thực hiện `new Address(...)` và tạm thời chúng ta sẽ cố định nó thành `support@example.com` và đặt tên là "support." Tiếp theo, chúng ta cần một chủ đề, chủ đề là "welcome," và sau đó chúng ta sẽ truyền vào các phần HTML và văn bản. Chúng ta cũng sẽ loại bỏ đoạn mã

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

vì chúng ta không gửi email ngay lập tức.

Không còn gửi email ngay lập tức, chúng ta chỉ đang xếp hàng nó. Điều đó có nghĩa là chúng ta không cần constructor nữa

```
public function __construct(protected MailerInterface $mailer)
{
}
```

vì vậy chúng ta có thể xóa nó. Hãy thử nghiệm điều này để đảm bảo rằng bản ghi được tạo trong cơ sở dữ liệu. Chúng ta sẽ mở trình duyệt

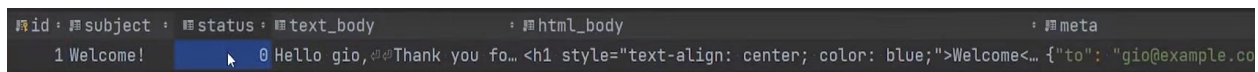


A registration form with a light blue background. It contains three input fields: 'Name' with the value 'gio', 'Email' with the value 'gio@example.com', and 'Password' with masked characters '.....'. Below the fields is a large blue button labeled 'Register' with a hand cursor icon pointing at it.

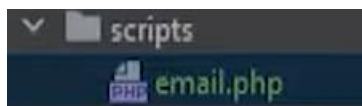
nhập một số thông tin (tên, email, mật khẩu), nhấn "register,". Chúng ta chuyển sang MailHog



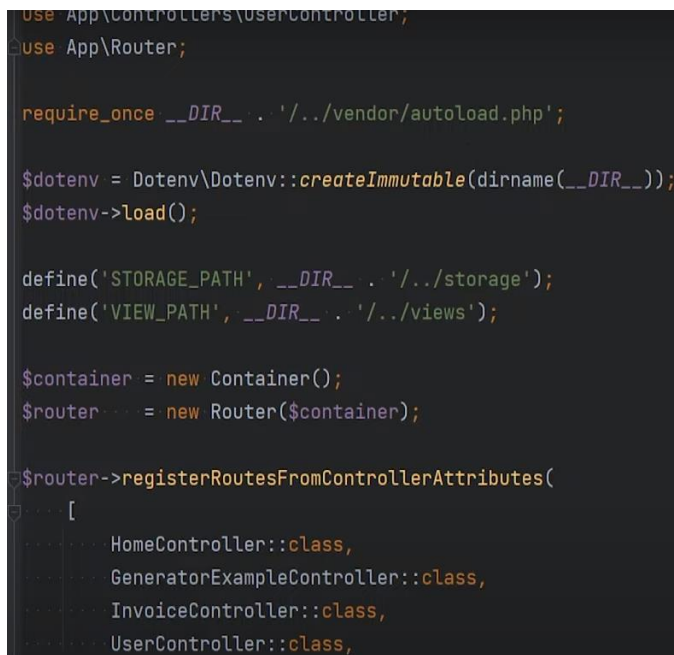
chúng ta thấy rằng không có email nào ở đây vì chúng ta chưa gửi email nào. Hãy mở mã nguồn, mở bảng, làm mới, và chúng ta thấy rằng bản ghi email đã được tạo. Chúng ta thấy trạng thái đã được đặt thành 0, có nghĩa là email đã được đưa vào hàng đợi.



Bây giờ, chúng ta cần tạo một số loại tập lệnh email sẽ chịu trách nhiệm gửi email. Chúng ta sẽ cần tạo một thư mục "scripts" và thêm một tệp script như "email.php" vào đó.



Tập lệnh này sẽ chạy trong dòng lệnh, không chạy trong trình duyệt, vì vậy chúng ta cần khởi động ứng dụng của chúng ta để truy cập cấu hình và các đối tượng hoặc phân tử cần thiết khác. Chúng ta cũng cần yêu cầu trình tải tự động (autoloader), vì vậy thực tế chúng ta cần thực hiện cùng một điều mà chúng ta đang làm trong tệp `public/index.php`.



Mã này sẽ chạy trong dòng lệnh, nên chúng ta không cần liên quan đến việc định tuyến (routing), vì vậy có thể loại bỏ bộ định tuyến và các phần không cần thiết khác. Chúng ta cần đối tượng container, nhưng không cần đường dẫn lưu trữ (storage path) hoặc đường dẫn xem (view path) ở đây.

```
define('STORAGE_PATH', __DIR__ . '/../storage');  
define('VIEW_PATH', __DIR__ . '/../views');
```

Bây giờ chúng ta cần làm việc với đối tượng ứng dụng của chúng ta

```
(new App(  
    $container,  
    $router,  
    [],  
    new Config($_ENV)  
))->run();
```

để làm cho nó hoạt động với tập lệnh của chúng ta, vì trong thời điểm hiện tại, nó sẽ không hoạt động vì router là một tham số bắt buộc.

Một tùy chọn là làm cho phần dependency của router là null. Chúng ta sẽ vào lớp `App`, sau đó thêm dòng import tại đầu tệp và đánh dấu tham số của router là null.

Sau đó, chúng ta sẽ sắp xếp lại một chút mã bằng cách di chuyển phần cấu hình lên phía trên, vì điều này là bắt buộc, và chúng ta sẽ đặt giá trị mặc định của request là một mảng trống vì cũng không cần thiết cho kịch bản dòng lệnh của chúng ta.

```
class App  
{  
    private static DB $db;  
  
    public function __construct(  
        protected Container $container,  
        protected Config $config,  
        protected ?Router $router = null,  
        protected array $request = [],  
    ) {
```

Có nhiều cách khác nhau để cấu trúc mã này, nhưng trong trường hợp này, chúng ta không đang xây dựng một framework thực sự nên không quan trọng lắm.

Một điều khác chúng ta muốn làm là di chuyển

```
$this->container->set(PaymentGatewayServiceInterface::class, PaymentGatewayService::class);  
$this->container->set(MailerInterface::class, fn() => new CustomMailer($config->mailer['dsn']));
```

ra và đặt nó vào một phương thức gọi là "boot". Chúng ta sẽ tạo một phương thức `boot` công khai ở đây để khởi động ứng dụng của chúng ta và chúng ta có thể thiết lập các binding và mọi thứ cần thiết trong đó. Chúng ta sẽ làm cho phương thức này trả về chính nó.

```
public function boot(): static  
{  
    $this->container->set(PaymentGatewayServiceInterface::class, PaymentGatewayService::class);  
    $this->container->set(MailerInterface::class, fn() => new CustomMailer($this->config->mailer['dsn']));  
    return $this;  
}
```

Nếu chúng ta muốn trở nên phức tạp hơn, thay vì chấp nhận cấu hình trong constructor, chúng ta có thể tạo cấu hình trong phương thức `boot`. Thay vì làm điều này trong cả `email.php` và `index.php`, chúng ta có thể di chuyển nó ra khỏi đó và đặt nó trong phương thức `boot`, sau đó chúng ta sẽ khởi tạo cấu hình ở đây.

```
public function boot(): static  
{  
    $dotenv = Dotenv::createImmutable(dirname(__DIR__));  
    $dotenv->load();  
    $this->config = new Config($_ENV);  
    $this->container->set(PaymentGatewayServiceInterface::class, PaymentGatewayService::class);
```

Sau đó, chúng ta có thể loại bỏ hoàn toàn sự phụ thuộc vào cấu hình và chỉ cần tạo thuộc tính `config` ở đây.

```
private Config $config;
```

Vì chúng ta đã thực hiện điều đó, chúng ta cũng cần di chuyển

```
) {  
    static::$db = new DB($config->db ?? []);  
}
```

ra vì nó cần truy cập vào cấu hình. Chúng ta sẽ lấy nó ra và đặt vào trong phương thức `boot`



```

public function boot(): static
{
    $dotenv = Dotenv::createImmutable(dirname(__DIR__));
    $dotenv->load();

    $this->config = new Config($_ENV);

    static::$db = new DB($config->db ?? []);

    $this->container->set(PaymentGatewayServiceInterface::class

```

sau đó truy cập config bằng cách sử dụng `\$this->config` và sau đó những gì chúng ta có thể làm là trong public index.php của chúng ta, chúng ta có thể loại bỏ

```

new Config($_ENV)

```

và chúng ta có thể loại bỏ hai dòng này

```

$dotenv = Dotenv\Dotenv::createImmutable(dirname(__DIR__));
$dotenv->load();

```

có nhiều cách khác nhau để thực hiện việc này, đây không phải là cách lý tưởng, đây không phải là cách hoàn hảo, chúng ta chỉ đang thử nghiệm mã nên bây giờ chúng tôi cần điều chỉnh public index.php của mình, chúng tôi cần gọi phương thức khởi động trước khi chạy

```

))->boot()->run();

```

Và sau đó, chúng ta sẽ thực hiện cùng điều đó trong tệp `email.php`. Chúng ta sẽ khởi tạo container của chúng ta và không cần truyền bất kỳ thứ gì ở đây. Thay vì gọi phương thức `run`, chúng ta sẽ đơn giản gọi phương thức `boot`.

```

$container = new Container();

(new App($container))->boot();

```

Vậy thay vì viết mã để thực sự gửi các email ở đây, chúng ta có thể tạo một loại lớp dịch vụ email để bao gồm logic đó. Chúng ta sẽ làm điều này bằng cách gọi phương thức `container get` để lấy đối tượng dịch vụ email và sau đó gọi phương thức `sendQueuedEmails`.

```
$container->get(\App\Services\EmailService::class)->sendQueuedEmails();
```

Phương thức `sendQueuedEmails` sẽ chịu trách nhiệm gửi tất cả các email trong hàng đợi.

Ở đây, chúng ta đang tạo một phương thức mới có tên `sendQueuedEmails` trong lớp dịch vụ email. Phương thức này sẽ lấy tất cả các email trong hàng đợi từ cơ sở dữ liệu và gửi chúng đi một cách tuần tự.

```
class EmailService
{
    public function __construct(protected \App\Models\Email $emailModel, protected \App\Interfaces\EmailInterface $mailer)
    {
    }

    public function sendQueuedEmails(): void
    {
        $emails = $this->emailModel->getEmailsByStatus(EmailStatus::Queue);

        foreach($emails as $email)
        {
        }
    }
}
```

Chúng ta đã thêm `emailModel` và `mailer` vào lớp này thông qua constructor, vì chúng ta cần sử dụng `emailModel` để truy vấn email trong hàng đợi và `mailer` để gửi email.

Trong phương thức `sendQueuedEmails`, chúng ta đã gọi phương thức `getEmailsByStatus` trên `emailModel` để lấy tất cả email trong hàng đợi. Phương thức `getEmailsByStatus` thực hiện một truy vấn đơn giản để lấy danh sách các email với trạng thái là "queue".

Sau đó, chúng ta đã lặp qua từng email trong danh sách và xây dựng đối tượng email tương ứng bằng cách sử dụng thông tin từ cơ sở dữ liệu.

Bây giờ chúng ta đã có một lớp dịch vụ email có khả năng lấy và gửi các email trong hàng đợi.

```
public function sendQueuedEmails(): void
{
    $emails = $this->emailModel->getEmailsByStatus(EmailStatus::Queue);

    foreach($emails as $email) {
        $meta = json_decode($email->meta, true);

        $emailMessage = (new Email())
            ->from($meta['from'])
            ->to($meta['to'])
            ->subject($email->subject)
            ->text($email->text_body)
            ->html($email->html_body);

        $this->mailer->send($emailMessage);

        $this->emailModel->markEmailSent($email->id);
    }
}
```

Chúng ta đã thêm các thông tin cần thiết để gửi email, bao gồm tiêu đề (subject), nội dung văn bản (text body), và nội dung HTML (html body) vào đối tượng email message.

Sau khi gửi email, chúng ta đã cập nhật trạng thái (status) của bản ghi email trong cơ sở dữ liệu thành "sent" và cập nhật thời gian gửi (sent at timestamp) bằng cách sử dụng phương thức `markEmailSent` trên đối tượng `emailModel`.

Điều này đảm bảo rằng các email được đánh dấu là "sent" trong cơ sở dữ liệu sau khi chúng đã được gửi đi và không bị gửi lại.

Phương thức `markEmailSent` đã được thiết lập để cập nhật bản ghi email trong bảng `emails` của cơ sở dữ liệu. Nó đặt trạng thái của email thành "sent" và cập nhật thời gian gửi (`sent\_at`) để ghi nhận rằng email đã được gửi đi thành công. Điều này giúp chúng ta theo dõi trạng thái của các email và tránh gửi chúng một lần nữa.

```
public function markEmailSent(int $id): void
{
    $stmt = $this->db->prepare(
        'UPDATE emails
        SET status = ?, sent_at = NOW()
        WHERE id = ?'
    );

    $stmt->execute([EmailStatus::Sent->value, $id]);
}
```

Chúng ta cũng có thể bọc phần này

```
public function sendQueuedEmails(): void
{
    $emails = $this->emailModel->getEmailsByStatus(EmailStatus::Queue);

    foreach($emails as $email) {
        $meta = json_decode($email->meta, true);

        $emailMessage = (new Email())
            ->from($meta['from'])
            ->to($meta['to'])
            ->subject($email->subject)
            ->text($email->text_body)
            ->html($email->html_body);

        $this->mailer->send($emailMessage);

        $this->emailModel->markEmailSent($email->id);
    }
}
```

vào trong một khối try catch, và nếu có điều gì đó sai khi gửi email, chúng ta có thể cập nhật trạng thái của bản ghi email trong bảng thành "failed" và có thể ghi lại ngoại lệ và nhiều hơn nữa. Vậy là chúng ta đã sẵn sàng, hãy thử nghiệm để đảm bảo rằng nó hoạt động. Điều tôi sẽ làm là chạy kịch bản email bằng tay từ terminal, vì vậy hãy mở terminal và chạy lệnh "php scripts/email.php".

```
root@1a7b39dd4c13:/var/www# php scripts/email.php
```

Chúng ta chờ một chút mở MailHog và có thể thấy email đã được gửi. Nếu chúng ta mở bảng dữ liệu ngay bây giờ, trạng thái này sẽ được cập nhật thành "sent", vì vậy nếu chúng ta làm mới, chúng ta sẽ thấy trạng thái đã được cập nhật thành "sent".

```
id : subject : status : text_body : ht... : meta
1 Welcome! 1 Hello gio, Thank you fo... <h1 sty... {"to": "gi
```

Chúng ta đã biết cách gửi email bằng cách chạy tập lệnh email.php. Tuy nhiên, vấn đề là phải chạy tập lệnh này thủ công. Làm thế nào để tự động hóa quá trình này để không phải thực hiện thủ công? Đó là nơi Cron có thể giúp ích. Cron là một công cụ lên lịch công việc dành cho các hệ thống Unix, cho phép chúng ta chạy các tập lệnh hoặc công việc tự động vào thời gian cụ thể hoặc theo lịch trình cụ thể. Lịch trình của các công việc hoặc tập lệnh được lưu trong một tệp gọi là "crontab," viết tắt của "cron table."

Mỗi mục hoặc dòng trong crontab đại diện cho công việc mà bạn muốn chạy và thực hiện. Một mục bao gồm sáu hoặc bảy trường, tùy thuộc vào cách triển khai cụ thể.

Trường đầu tiên: Phút (0 đến 59)

Trường thứ hai: Giờ (0 đến 23)

Trường thứ ba: Ngày trong tháng (1 đến 31)

Trường thứ tư: Tháng trong năm (1 đến 12)

Trường thứ năm: Ngày trong tuần (0 đến 6, với 0 là Chủ Nhật)

Trường thứ sáu: Lệnh hoặc tập lệnh cần thực thi

Chúng ta có thể xác định nhiều giá trị cho các trường bằng cách sử dụng dấu phẩy, ví dụ: 1,15,30 \* \* \* \* /đường\_dẫn/tới/tập\_lệnh sẽ chạy tập lệnh vào các phút 1, 15 và 30 của mỗi giờ.

Nếu chúng ta muốn chạy tập lệnh theo khoảng thời gian cố định, bạn có thể sử dụng một ký tự chia bằng dấu gạch chéo sau dấu sao, ví dụ: \*/2 \* \* \* \* /đường\_dẫn/tới/tập\_lệnh sẽ chạy tập lệnh mỗi 2 phút.

Sau khi đã có biểu thức Cron, chúng ta cần thêm nó vào tệp crontab. Trên các hệ thống Linux thông thường, chúng ta có thể mở tệp crontab bằng cách chạy lệnh crontab -e, sau đó thêm mục của chúng ta vào tệp và lưu lại. Trên các hệ thống khác, chúng ta có thể cần chỉnh sửa tệp crontab trực tiếp bằng cách chỉnh sửa tệp /etc/crontab.

Tuy nhiên, chúng ta đang làm việc trong một container Docker, vì vậy chúng ta cần chạy cron trong container của chúng ta. Có một số cách để làm điều này, một trong số đó là chạy cron trong một container riêng biệt, và cách khác là chạy nó trong cùng một container với ứng dụng của chúng ta. Chúng ta sẽ chạy nó trong một container riêng biệt.

```
cron:
  container_name: programwithgio-cron
  build: ./cron
  volumes:
    - ../:/var/www
    - ./log/cron:/var/log/cron
```

Bây giờ, hãy mở tệp Docker thực tế trong thư mục `chrome`. Nếu chúng ta mở dự án và mở thư mục `docker`, chúng ta sẽ thấy trong thư mục `chrome` tôi có tệp Docker và tệp crontab.



Vậy, chúng ta hãy mở tệp Docker.

```
FROM php:8.1-fpm-alpine

RUN docker-php-ext-install pdo pdo_mysql

COPY crontab /etc/crontabs/root

RUN mkdir /var/log/cron

CMD ["crond", "-f"]
```

Tệp Docker này đang tải từ hình ảnh php 8.1 fpm alpine. Chúng ta sử dụng Alpine vì nó đã có cài đặt cron sẵn. Vì vậy, điều duy nhất chúng ta cần thực hiện là sao chép tệp crontab cục bộ của chúng ta vào container, và đó chính là những gì chúng ta đang làm. Chúng ta đang sao chép tệp



crontab vào `/etc/crontabs/root`, sau đó chúng ta tạo một thư mục để lưu trữ các tệp nhật ký của cron, và cuối cùng, chúng ta chạy lệnh cron.

Chúng ta hãy mở tệp crontab, đây là nơi chúng ta đặt biểu thức cron.

```
*/2 * * * * php /var/www/scripts/email.php >> /var/log/cron/cron.log 2>&1
```

Đây chính là biểu thức cron mà chúng ta vừa thấy, sau đó chúng ta đang đưa toàn bộ đầu ra vào tệp nhật ký cron. Bây giờ, tôi sẽ thoát khỏi đây và chạy lệnh:

```
root@1a7b39dd4c13:/var/www# exit
exit

gge1a@DESKTOP-45V9RT2 MINGW64 /d/code/learnphp
$ docker-compose up -d --build
```

Để khởi động lại các container Docker và xây dựng lại chúng.

Và trong khi nó đang làm việc của nó, chúng ta hãy mở trình duyệt và làm mới trang web bằng cách nhấn nút F5 hoặc Refresh. Chúng ta đã gửi biểu mẫu với các giá trị giống nhau, vì vậy nó sẽ tạo một bản ghi email trong bảng cơ sở dữ liệu để xếp hàng email, và nếu chúng ta làm mới bảng emails, chúng ta sẽ thấy có bản ghi thứ hai ở đó.

id	subject	status	text_body	ht...	meta
1	Welcome!	1	Hello gio, Thank you fo...	<h1 sty...	{"to": "gio@examp
2	Welcome!	0	Hello gio, Thank you fo...	<h1 sty...	{"to": "gio@examp

Bây giờ, công việc của cron được dự định chạy mỗi phút cách nhau, vì vậy trạng thái của bản ghi email đã thay đổi thành "1". Điều này có nghĩa là bản ghi email đã được gửi thành công.

id	subject	status	text_body	ht...	meta
1	Welcome!	1	Hello gio, Thank you fo...	<h1 sty...	{"to": "gio@exa
2	Welcome!	1	Hello gio, Thank you fo...	<h1 sty...	{"to": "gio@exa