Cours Laravel 5.5 – les bases – envoyer un email

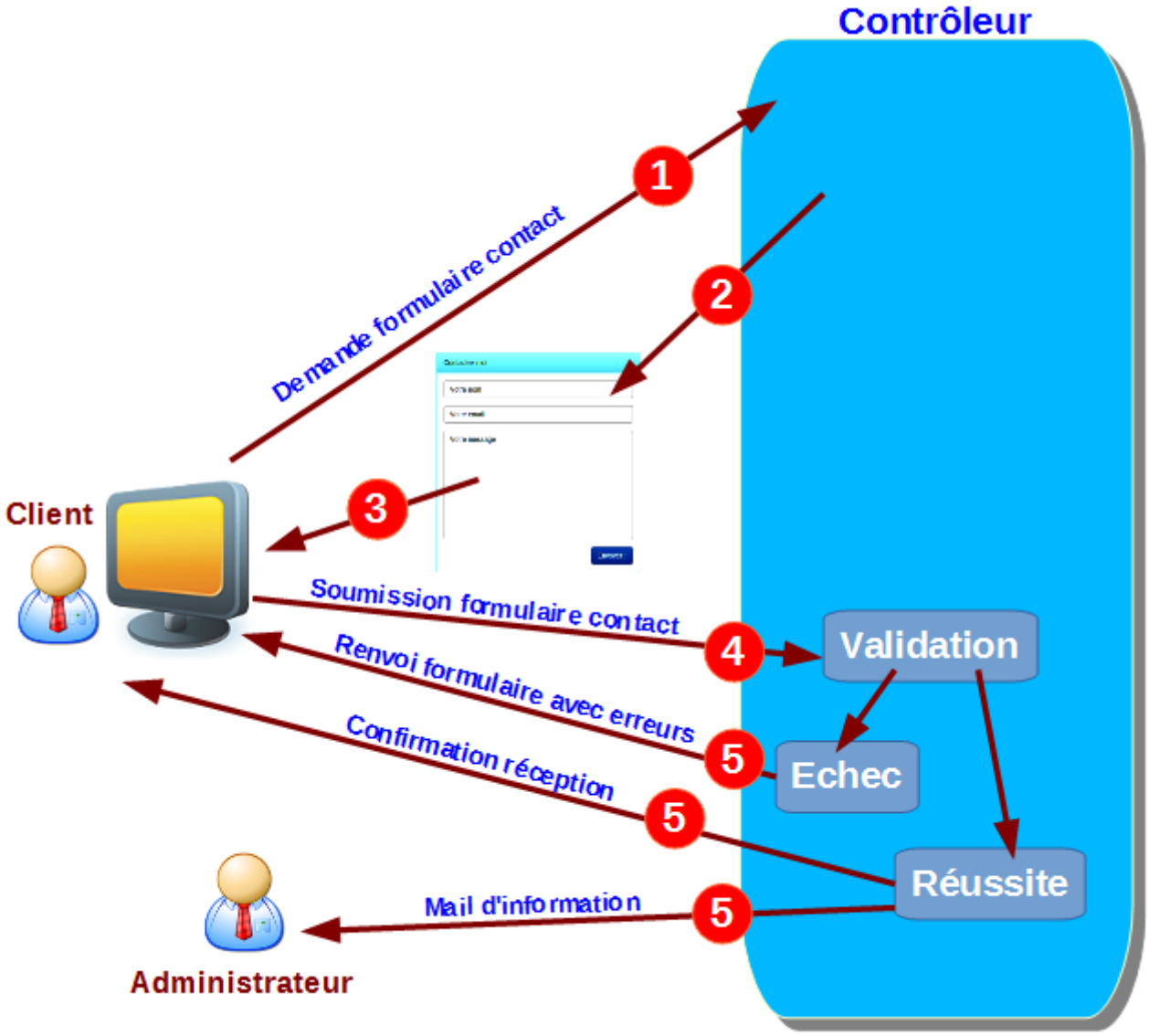
Laravel utilise le célèbre composant SwiftMailer pour l'envoi des emails. Mais il en simplifie grandement l'utilisation. Dans ce chapitre nous allons prolonger l'exemple précédent de la prise de contact en ajoutant l'envoi d'un email à l'administrateur du site lorsque quelqu'un soumet une demande de contact.

On va donc prendre le code tel qu'on l'a laissé lors du précédent chapitre et le compléter en conséquence.

On verra plus tard que Laravel propose aussi un système complet de notification qui permet entre autres l'envoi d'emails.

Le scénario

Le scénario est donc le même que pour le précédent chapitre avec l'ajout d'une action :



On va avoir les mêmes routes et vues, c'est uniquement au niveau du contrôleur que le code va évoluer pour intégrer cette action complémentaire.

Configuration

Si vous regardez dans le fichier **.env** vous trouvez une section qui concerne les emails :

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=null
MAIL_PASSWORD=null
MAIL_ENCRYPTION=null
```

Les valeurs correspondent au prestataire utilisé.

Au niveau du driver, le plus classique est certainement le SMTP mais vous pouvez aussi utiliser mail, mailgun (gratuit jusqu'à 10 000 envois par mois), ses, sparkpost…

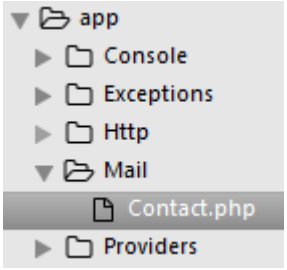
Vous devez correctement renseigner les paramètres pour que ça fonctionne selon votre contexte (en local avec le SMTP de votre prestataire, en production avec la fonction mail de PHP ou d'un autre système…).

La classe Mailable

Avec Laravel, pour envoyer un email il faut passer par la création d'une classe « mailable ». Encore une fois c'est Artisan qui va nous permettre de créer notre classe :

```
php artisan make:mail Contact
```

Comme le dossier n'existe pas il est créé en même temps que le fichier de la classe :



Par défaut on a ce code :

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class Contact extends Mailable
{
```

```
    use Queueable, SerializesModels;

    /**
     * Create a new message instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->view('view.name');
    }
}
```

Tout se passe dans la méthode **build**. On voit qu'on retourne une vue, celle-ci doit comporter le code pour le contenu de l'email.

On commence par changer le nom de la vue :

```
return $this->view('emails.contact');
```

On va créer cette vue (**resources/views/emails/contact.blade.php**) :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
  </head>
  <body>
    <h2>Prise de contact sur mon beau site</h2>
    <p>Réception d'une prise de contact avec les éléments suivants
:</p>
    <ul>
      <li><strong>Nom</strong> : {{ $contact['nom'] }}</li>
      <li><strong>Email</strong> : {{ $contact['email'] }}</li>
```

```
        <li><strong>Message</strong> : {{ $contact['message'] }}</li>
    </ul>
  </body>
</html>
```

Pour que ça fonctionne on doit transmettre à cette vue les entrées de l'utilisateur. Il faut donc passer les informations à la classe **Contact** à partir du contrôleur.

On peut aussi créer des email en Markdown.

Transmission des informations à la vue

Il y a deux façons de procéder pour transmettre les informations, nous allons utiliser la plus simple. Il suffit de créer une propriété obligatoirement publique dans la classe « mailable » et celle-ci sera automatiquement transmise à la vue. Voici le nouveau code de notre classe :

```
<?php

namespace App\Mail;

use Illuminate\Bus\Queueable;
use Illuminate\Mail\Mailable;
use Illuminate\Queue\SerializesModels;
use Illuminate\Contracts\Queue\ShouldQueue;

class Contact extends Mailable
{
    use Queueable, SerializesModels;

    /**
     * Elements de contact
     * @var array
     */
    public $contact;

    /**
```

```
     * Create a new message instance.
     *
     * @return void
     */
    public function __construct(Array $contact)
    {
        $this->contact = $contact;
    }

    /**
     * Build the message.
     *
     * @return $this
     */
    public function build()
    {
        return $this->from('monsite@chezmoi.com')
            ->view('emails.contact');
    }
}
```

J'en ai aussi profité pour préciser l'adresse de l'expéditeur avec la méthode **from**. On pourrait attacher un document avec **attach**, faire une copie avec **cc…**

J'ai créé la propriété publique **$contact** qui sera renseignée par l'intermédiaire du constructeur.

Il ne reste plus qu'à modifier le contrôleur pour envoyer cet email avec les données nécessaires :

```
<?php

namespace App\Http\Controllers;

use App\Http\Requests\ContactRequest;
use Illuminate\Support\Facades\Mail;
use App\Mail\Contact;

class ContactController extends Controller
{
    public function create()
    {
        return view('contact');
```

```
    }

    public function store(ContactRequest $request)
    {
        Mail::to('administrateur@chezmoi.com')
            ->send(new Contact($request->except('_token')));

        return view('confirm');
    }
}
```

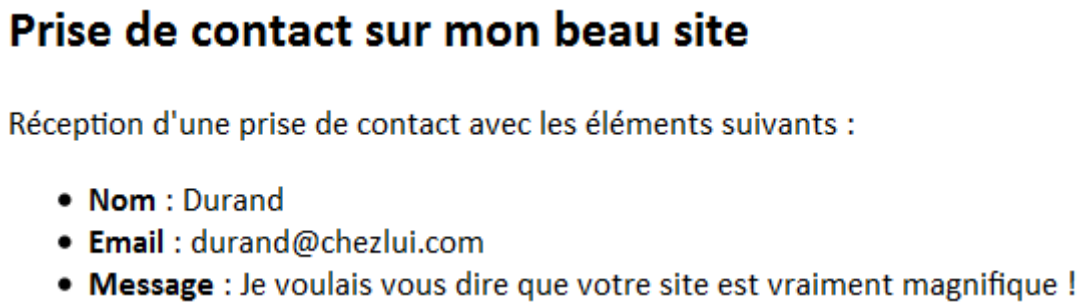
Tout se passe sur cette ligne :

```
Mail::to('administrateur@chezmoi.com')
    ->send(new Contact($request->except('_token')));
```

On a :

- l'adresse de l'administrateur (to),
- l'envoi avec la méthode send,
- la création d'une instance de la classe Contact avec la transmission des données saisies (mis à part le token pour la protection CSRF qui ne sert pas).

Et si tout se passe bien le message doit arriver jusqu'à l'administrateur :



Comme l'envoi d'emails peut prendre beaucoup de temps on utilise en général une file d'attente (queue). Il faut changer ainsi le code dans le contrôleur :

```
Mail::to('administrateur@chezmoi.com')
            ->queue(new Contact($request->except('_token')));
```

Mais évidemment pour que ça fonctionne il faut avoir paramétré et lancé un système de file d'attente. J'en parlerai sans doute dans

un chapitre ultérieur.

Test de l'email

Quand on crée la vue pour l'email il est intéressant de voir l'aspect final avant de faire un envoi réel. Pour le faire il suffit de créer une simple route :

```
Route::get('/test-contact', function () {
    return new App\Mail\Contact([
        'nom' => 'Durand',
        'email' => 'durand@chezlui.com',
         'message' => 'Je voulais vous dire que votre site est magnifique !'
        ]);
});
```

Maintenant en utilisant l'url **monsite/test-contact** on obtient l'aperçu directement dans le navigateur !

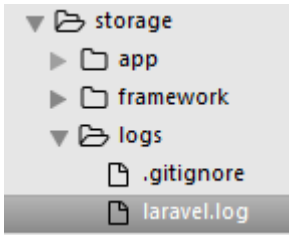
Envoyer des emails en phase développement

Le mode Log

Lorsqu'on est en phase de développement il n'est pas forcément pratique ou judicieux d'envoyer réellement des emails. Un solution simple consiste à passer en mode Log en renseignant le driver dans le fichier **.env** :

```
MAIL_DRIVER=log
```

Les emails ne seront pas envoyés mais le code en sera mémorisé dans le fichier des logs :



Vous allez y trouver par exemple ce contenu :

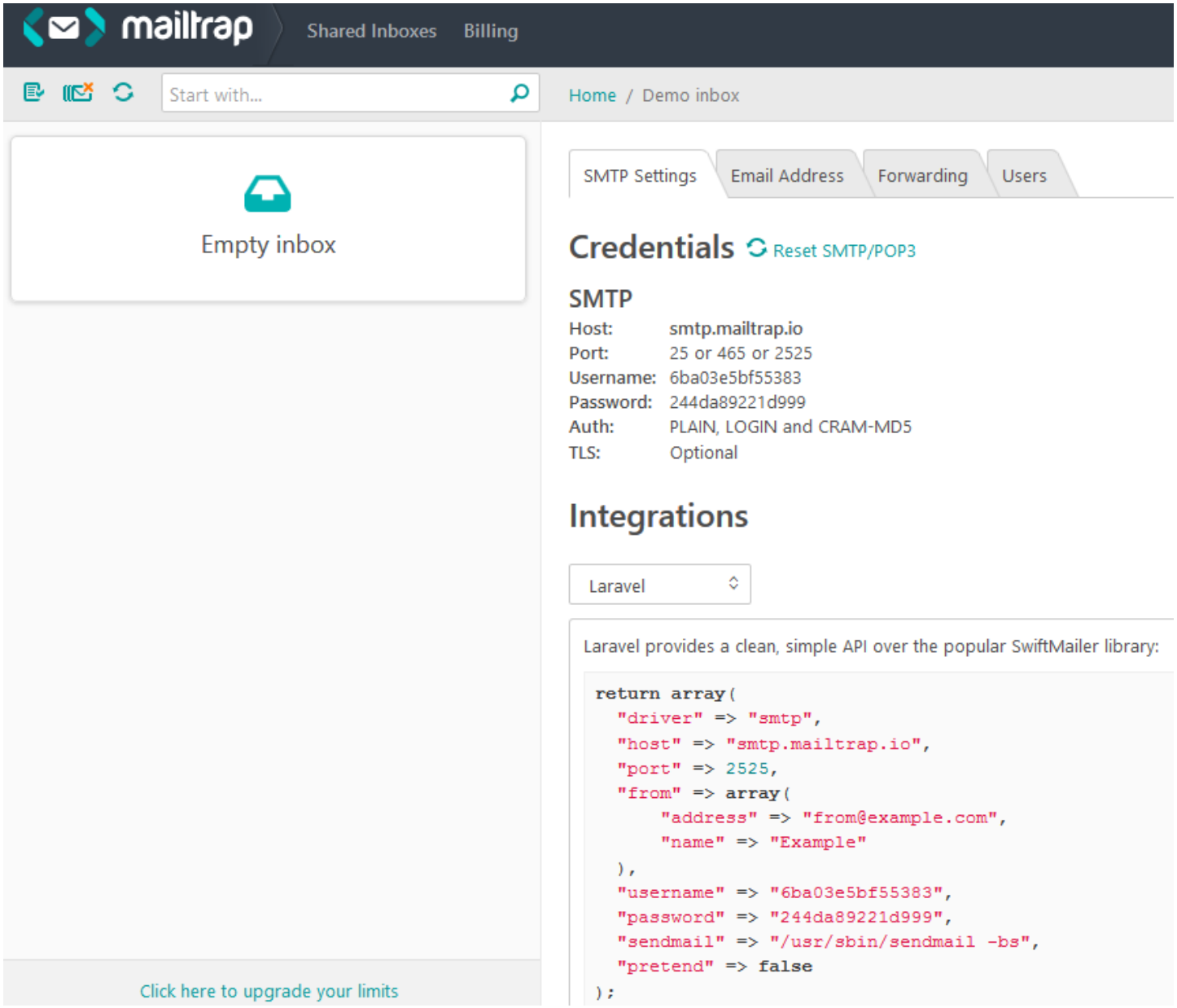
```
[2017-09-10    14:18:13]    local.DEBUG:    Message-ID:
<b25ad1d634f7ca660d1d7bbf81a8463a@laravel5.dev>
Date: Sun, 10 Sep 2017 14:18:13 +0000
Subject: Contact
From: monsite@chezmoi.com
To: administrateur@chezmoi.com
MIME-Version: 1.0
Content-Type: text/html; charset=utf-8
Content-Transfer-Encoding: quoted-printable

<!DOCTYPE html>
<html lang="fr">
 <head>
 <meta charset="utf-8">
 </head>
 <body>
 <h2>Prise de contact sur mon beau site</h2>
 <p>Réception d'une prise de contact avec les éléments suivants
:</p>
 <ul>
 <li><strong>Nom</strong> : Durand</li>
 <li><strong>Email</strong> : durand@chezlui.fr</li>
 <li><strong>Message</strong> : Je voulais vous dire que votre
site est magnifique !</li>
 </ul>
 </body>
</html>
```

Ce qui vous permet de vérifier que tout se passe correctement (hormis l'envoi).

MailTrap

Une autre possibilité très utilisée est MailTrap qui a une option gratuite (une boîte, avec 50 messages au maximum et 2 messages par

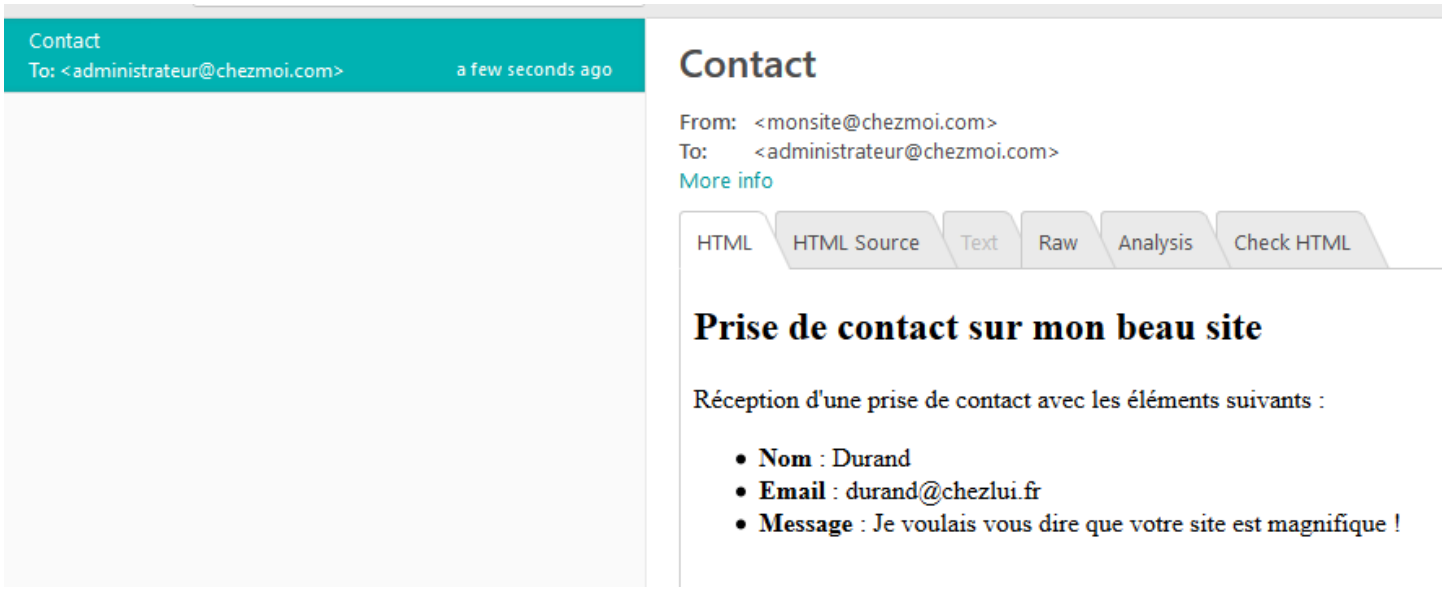
seconde). Vous avez un tableau de bord et une boîte de messages :



J’ai fait apparaître les configurations pour Laravel. Il est ainsi facile de renseigner le fichier **.env** :

```
MAIL_DRIVER=smtp
MAIL_HOST=smtp.mailtrap.io
MAIL_PORT=2525
MAIL_USERNAME=6ba03e5bf55383
MAIL_PASSWORD=244da89221d999
MAIL_ENCRYPTION=null
```

Avec cette configuration lorsque j’envoie un email je le vois arriver :



Vous pouvez analyser l'email avec précision, je vous laisse découvrir toutes les options.

En résumé

- Laravel permet l'envoi simple d'email.
- Il faut configurer correctement les paramètres pour l'envoi des emails.
- Pour chaque email il faut créer une classe « mailable ».
- On peut passer des informations à l'email en créant une propriété publique dans la classe « mailable ».
- On peut passer en mode Log en phase de développement ou alors utiliser MailTrap.