Laravel 4 : chapitre 35 : Sentry 3

*Le projet Sentry 3 sur lequel je me suis basé pour ce tuto a été renommé **Sentinel** et est devenu payant. Cet article est donc devenu automatiquement obsolète. Je le laisse tout de même pendant quelques temps parce que le code peut encore intéresser…*

Le package le plus utilisé avec Laravel est certainement Sentry, un système d'authentification simple et performant. On en arrive d'ailleurs à se demander pourquoi il n'a pas été intégré directement comme système d'authentification, mais ça viendra peut-être. J'avais commencé à rédiger un article concernant la version 2 mais comme la 3 est sur les rails et commence à se stabiliser j'ai préféré traiter de cette nouvelle version. Évidemment il se pourrait que des modifications en cours du développement rendent certains éléments exposés ici obsolètes, je m'efforcerai de garder cet article à jour avec les évolutions.

Attention ! Sentry nécessite au minimum la version 5.4 de PHP !

Qu'apporte Sentry par rapport au système d'authentification de base de Laravel que nous avons eu l'occasion de voir ? Essentiellement :

- activation des utilisateurs,
- permissions par utilisateur,
- création de groupes avec permissions,
- throttling (je n'ai pas trouvé de traduction adéquate) qui est une sécurisation en cas de tentatives de connexion répétées.

Pour cet article je vous propose un exemple simple de mise en œuvre sans création de groupes mais en utilisant l'activation, les permissions, la connexion, la déconnexion, la réinitialisation du mot de passe et la gestion des utilisateurs.

Pour faire bonne mesure je vais aussi utiliser la possibilité de traduction offerte par Laravel que je n'ai pas encore eu l'occasion de traiter. Je vais aussi prévoir l'utilisation d'Ajax pour la déconnexion et la suppression d'un utilisateur.

Installation

Pour vous éviter des manipulations laborieuses j'ai réuni le code dans un fichier compressé à télécharger ici. Il suffit de le décompresser dans un dossier et de lancer l'installation :

```
composer install
```

Créez ensuite une base et configurez-la correctement dans le fichier app/config/database.php.

Pour l'envoi des Email configurez aussi le fichier app/config/mail.php.

Enfin lancez les migrations :

```
php artisan migrate --package="cartalyst/sentry"
```

Il faut ensuite effectuer la population pour créer les 3 utilisateurs prévus pour tester l'application :

```
php artisan db:seed
```

Si vous regardez le code pour cette population vous verrez que j'ai utilisé la méthode registerAndActivate de Sentry, par exemple pour l'administrateur :

```
Sentry::registerAndActivate(array(
        'email' => 'admin@appli.com',
        'password' => 'password',
        'first_name' => 'Jean',
        'last_name' => 'Passe',
        'permissions' => array(
                'admin' => true
        )
));
```

Au niveau des permissions j'ai défini trois types :

- admin,
- moderator,
- user.

Pour terminer il est judicieux de publier le fichier de configuration de Sentry dans votre application (ça vous permettra d'apporter des modifications sans les perdre) :

```
php artisan config:publish cartalyst/sentry
```

Vous devriez normalement avoir une application fonctionnelle.

Si vous regardez le fichier composer.json vous allez voir que j'ai prévu de charger la version de Sentry en cours de développement :

```
{
   ...
   "require": {
        "laravel/framework": "4.1.*",
        "cartalyst/sentry": "3.0.*@dev"
   },
   ...
}
```

Vous trouverez aussi un chargement de classes à la mode PSR-0 pour le dossier app/lib :

```
"psr-0": {
        "Lib": "app"
}
```

Ce dossier contient tout le code de 'lapplication.

Pour le fonctionnement de Sentry le service est déclaré dans app/config/app.php :

```
'Cartalyst\Sentry\Laravel\SentryServiceProvider',
```

Il y a également la déclaration des 4 façades (on en avait qu'une avec Sentry 2) dans le même fichier :

```
'Activation'                                                    =>
'Cartalyst\Sentry\Laravel\Facades\Activation',
'Reminder'          => 'Cartalyst\Sentry\Laravel\Facades\Reminder',
'Sentry'            => 'Cartalyst\Sentry\Laravel\Facades\Sentry',
```

```
'SwipeIdentity'                                                    =>
'Cartalyst\Sentry\Laravel\Facades\SwipeIdentity',
```

Dans l'exemple on n'utilisera pas SwipeIdentity, vous pouvez donc ne pas le mentionner.

Organisation du code

J'ai fini par prendre des habitudes au niveau de l'organisation du code et désormais je n'utilise plus les dossiers controllers et views et je crée un dossier app/lib chargé de contenir tout le code de l'application organisé en espaces de nommage conformément au PSR-0. Pour indiquer à Laravel que les vues se trouvent dans ce dossier il faut l'informer dans le fichier app/config/view.php :

```
'paths' => array(__DIR__.'/../Lib'),
```

Pour les contrôleurs pas de souci il les trouvera !

Les routes

J'aime bien avoir un fichier de routes simple et lisible. Pour y arriver j'utilise autant que possible des contrôleurs de ressource et RESTful. Pour cette application on va avoir ce fichier de routage app/routes.php :

```
Route::get('/', array('as' => 'home', function()
{
  return View::make('Shared.views.home');
}));

Route::group(array('before' => 'uncheck'), function()
{
                                Route::controller('login',
'Lib\User\Login\LoginController');
                            Route::controller('register',
'Lib\User\Register\RegisterController');
                                Route::controller('reset',
'Lib\User\Reset\ResetController');
       Route::controller('pass', 'Lib\User\Pass\PassController');
});
```

```
Route::group(array('before' => 'check'), function()
{
                                Route::controller('logout',
'Lib\User\Logout\LogoutController');
                               Route::controller('account',
'Lib\User\Account\AccountController');
});

Route::group(array('before' => 'access:admin'), function()
{
       Route::resource('user', 'Lib\Admin\User\UserController');
});
```

Peu de lignes mais plein de routes comme vous pouvez le vérifier avec Artisan (php artisan routes) :

```
+-----------------------------------------------------------------
----+-------------+-----------------------------------------------
------+
|                                                                 URI
|    Name                                                  |    Action
|
+-----------------------------------------------------------------
----+-------------+-----------------------------------------------
------+
|                         GET|HEAD                                 /
|    home                                                  |    Closure
|
|  GET|HEAD  login/index/{one?}/{two?}/{three?}/{four?}/{five?}
|                | Lib\User\Login\LoginController@getIndex
|
|                      GET|HEAD                               login
|                | Lib\User\Login\LoginController@getIndex
|
|   POST   login/index/{one?}/{two?}/{three?}/{four?}/{five?}
|                | Lib\User\Login\LoginController@postIndex
|
|                         POST                               login
|                | Lib\User\Login\LoginController@postIndex
|
|    GET|HEAD|POST|PUT|PATCH|DELETE    login/{_missing}
|              | Lib\User\Login\LoginController@missingMethod
|
|   PUT   logout/index/{one?}/{two?}/{three?}/{four?}/{five?}
```

```
|                | Lib\User\Logout\LogoutController@putIndex
|
|                          PUT                              logout
|                | Lib\User\Logout\LogoutController@putIndex
|
|    GET|HEAD|POST|PUT|PATCH|DELETE    logout/{_missing}
|              | Lib\User\Logout\LogoutController@missingMethod
|
|  GET|HEAD  account/index/{one?}/{two?}/{three?}/{four?}/{five?}
|                | Lib\User\Account\AccountController@getIndex
|
|                       GET|HEAD                        account
|                | Lib\User\Account\AccountController@getIndex
|
|   POST  account/profil/{one?}/{two?}/{three?}/{four?}/{five?}
|              | Lib\User\Account\AccountController@postProfil
|
|  POST  account/password/{one?}/{two?}/{three?}/{four?}/{five?}
|             | Lib\User\Account\AccountController@postPassword
|
|    GET|HEAD|POST|PUT|PATCH|DELETE    account/{_missing}
|             | Lib\User\Account\AccountController@missingMethod
|
|  GET|HEAD  register/index/{one?}/{two?}/{three?}/{four?}/{five?}
|                | Lib\User\Register\RegisterController@getIndex
|
|                       GET|HEAD                       register
|              | Lib\User\Register\RegisterController@getIndex
|
|   POST  register/index/{one?}/{two?}/{three?}/{four?}/{five?}
|              | Lib\User\Register\RegisterController@postIndex
|
|                         POST                           register
|              | Lib\User\Register\RegisterController@postIndex
|
|                                                    GET|HEAD
register/activation/{one?}/{two?}/{three?}/{four?}/{five?}    |
|  Lib\User\Register\RegisterController@getActivation |
|    GET|HEAD|POST|PUT|PATCH|DELETE    register/{_missing}
|                                                           |
Lib\User\Register\RegisterController@missingMethod |
|  GET|HEAD  reset/index/{one?}/{two?}/{three?}/{four?}/{five?}
|                | Lib\User\Reset\ResetController@getIndex
```

```
|
|                     GET|HEAD                          reset
|              | Lib\User\Reset\ResetController@getIndex
|
|   POST   reset/index/{one?}/{two?}/{three?}/{four?}/{five?}
|              | Lib\User\Reset\ResetController@postIndex
|
|                        POST                          reset
|              | Lib\User\Reset\ResetController@postIndex
|
|    GET|HEAD|POST|PUT|PATCH|DELETE    reset/{_missing}
|             | Lib\User\Reset\ResetController@missingMethod
|
|  GET|HEAD  pass/index/{one?}/{two?}/{three?}/{four?}/{five?}
|               | Lib\User\Pass\PassController@getIndex
|
|                     GET|HEAD                          pass
|               | Lib\User\Pass\PassController@getIndex
|
|   POST   pass/index/{one?}/{two?}/{three?}/{four?}/{five?}
|               | Lib\User\Pass\PassController@postIndex
|
|                        POST                          pass
|               | Lib\User\Pass\PassController@postIndex
|
|    GET|HEAD|POST|PUT|PATCH|DELETE    pass/{_missing}
|             | Lib\User\Pass\PassController@missingMethod
|
|                     GET|HEAD                          user
|  user.index      |  Lib\Admin\User\UserController@index
|
|                   GET|HEAD                    user/create
|  user.create    |  Lib\Admin\User\UserController@create
|
|                        POST                          user
|  user.store      |  Lib\Admin\User\UserController@store
|
|                   GET|HEAD                    user/{user}
|  user.show       |  Lib\Admin\User\UserController@show
|
|               GET|HEAD                user/{user}/edit
|  user.edit       |  Lib\Admin\User\UserController@edit
|
```

```
|                      PUT                          user/{user}
|  user.update    |  Lib\Admin\User\UserController@update
|
|                     PATCH                         user/{user}
|                | Lib\Admin\User\UserController@update
|
|                    DELETE                         user/{user}
|  user.destroy   |  Lib\Admin\User\UserController@destroy
|
+-------------------------------------------------------------
----+-------------+-------------------------------------------
------+
```

On verra comment sont constitués les filtres un peu plus loin.

Il existe des critiques concernant l'utilisation de ces routes « magiques », par exemple celle de Phil Sturgeon dans le chapitre « Restful Controllers ». Sa principale remarque concerne l'absence de nommage des routes. Il est pourtant parfaitement possible de nommer les routes d'un contrôleur RESTful en utilisant un troisième paramètre. Par exemple :

```
Route::controller('login',   'Lib\User\Login\LoginController',
array('getIndex' => 'login'));
```

On peut vérifier avec Artisan :

```
GET|HEAD    login/index/{one?}/{two?}/{three?}/{four?}/{five?}
| login          | Lib\User\Login\LoginController@getIndex
```

On a bien maintenant une route nommée login.

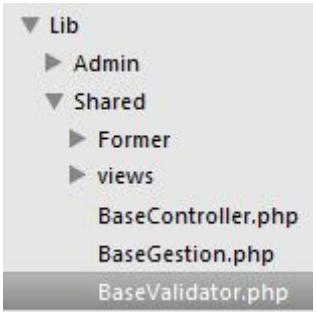
La validation

Pour la validation j'utilise cette classe de base :

```
<?php namespace Lib\Shared;

use Validator, Input;

abstract class BaseValidator {

        protected $regles = array();
        protected $errors = array();
```

```
        protected $messages = array();

        public function fails($id = null)
        {
                        if(!is_null($id)) $this->regles =
str_replace('id', $id, $this->regles);

                  $validation = Validator::make(Input::all(),
$this->regles, $this->messages);

                if($validation->fails())
                {
                        $this->errors = $validation->messages();
                        return true;
                }
                return false;
        }

        public function errors()
        {
                return $this->errors;
        }

}
```

Ce qui donne cette localisation dans les dossiers :

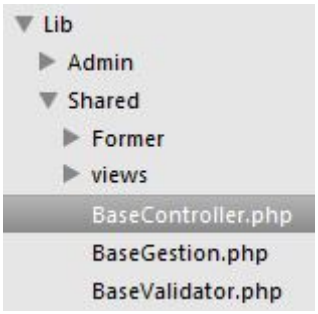


Après je n’ai plus qu’à prévoir des classes légères contenant les règles et les messages éventuels.

Contrôleurs et gestion

Dans l’application les contrôleurs sont dans leurs dossiers fonctionnels, j’ai prévu un contrôleur de base avec juste le filtre CSRF pour le verbe POST :

```
<?php namespace Lib\Shared;

abstract class BaseController extends \Illuminate\Routing\Controller {

	public function __construct()
	{
		$this->beforeFilter('csrf', array('on' => 'post'));
	}

}
```



Comme je limite l'utilisation des contrôleurs à la réception des requêtes et à l'émission des réponses j'ai besoin de classes de gestion. Celles-ci sont également dans leurs dossiers fonctionnels. J'ai aussi prévu une classe de base :

```
<?php namespace Lib\Shared;

use Sentry;

abstract class BaseGestion
{

	public function check()
	{
		return Sentry::check();
	}

	public function getUser()
	{
		return Sentry::getUser();
	}

}
```

Avec ces deux méthodes :

- **check** : pour vérifier si l'utilisateur encours est authentifié,
- **getUser** : pour récupérer les informations de l'utilisateur en cours.

Avec la méthode check il est facile de créer les filtres de base :

```
Route::filter('check', function()
{
	if (!Sentry::check()) return Redirect::guest('login');
});

Route::filter('uncheck', function()
{
	if (Sentry::check()) return Redirect::to('/');
});
```

On verra le filtre access un peu plus loin.

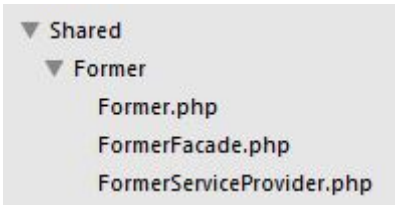
Former

Pour simplifier l'écriture des formulaires j'ai prévu un service pour les principaux contrôles utilisés. Par exemple voici la méthode générique pour les contrôles classiques :

```
public function control($type, $errors, $nom, $label, $valeur = '', $placeholder = '')
{
	if(Request::old($nom)) $valeur = Request::old($nom);
	   $attributes = array('class' => 'form-control', 'placeholder' => $placeholder);
	return sprintf('
		<div class="form-group %s">
```

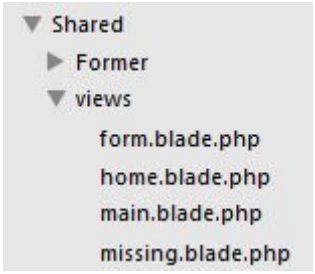
```
                %s
                %s
                <small class="text-danger">%s</small>
                </div>',
                $errors->has($nom) ? 'has-error' : '',
                parent::label($nom, $label),
                   call_user_func_array(array($this, $type),
array($nom, $valeur, $attributes)),
                $errors->first($nom)
        );
}
```

Je ne détaille pas cette partie dans cet article. Au niveau des dossiers on trouve cette partie dans 3 fichiers :



Vues communes

Il y a 4 vues communes :



On a un template main.blade.php pour le site :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
     <meta name="viewport" content="width=device-width, initial-
scale=1">
    <title>{{ trans('main.mon_joli_site') }}</title>
                                                          {{
HTML::style('https://netdna.bootstrapcdn.com/bootstrap/3.1.1/css/b
```

```
ootstrap.min.css') }}
                                                                {{
HTML::style('https://netdna.bootstrapcdn.com/bootstrap/3.1.1/css/b
ootstrap-theme.min.css') }}
    <!--[if lt IE 9]>
                                                                {{
HTML::style('https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv
.js') }}
                                                                {{
HTML::style('https://oss.maxcdn.com/libs/respond.js/1.4.2/respond.
min.js') }}
    <![endif]-->
  </head>
  <body>
    <nav class="navbar navbar-inverse" role="navigation">
      <div class="container">
        <div class="navbar-header">
          <a class="navbar-brand" href="{{ url('/') }}">Mon joli
site</a>
        </div>
        <ul class="nav navbar-nav">
          @if(\Sentry::check())
             <li{{ Request::is('account') ? ' class="active"' :
null      }}      id="compte">{{      link_to('account',
trans('main.mon_compte')) }}</li>
                <li id="deconnexion">{{ link_to('logout',
trans('main.deconnexion')) }}</li>
            @if(\Sentry::hasAccess('admin'))
               <li{{ Request::is('user*') ? ' class="active"' :
null   }}   id="admin">{{   link_to_route('user.index',
trans('main.admin')) }}</li>
            @endif
          @endif
          <li id="connexion"{{ \Sentry::check()? ' class="hidden"'
: null }} {{ Request::is('login') ? ' class="active"' : null }}>{{
link_to('login', trans('main.connexion')) }}</li>
        </ul>
      </div>
    </nav>
    <div class="container">
      @yield('contenu')
    </div>
                                                                {{
```

```
HTML::script('http://ajax.googleapis.com/ajax/libs/jquery/1.11.0/jquery.min.js') }}
                                                                    {{
HTML::script('http://netdna.bootstrapcdn.com/bootstrap/3.1.1/js/bootstrap.min.js') }}
    @if(\Sentry::check())
      <script>
        $(function(){
          $('#deconnexion a').click(function(e){
            e.preventDefault();
            $.ajax({
              url: 'logout',
              type: 'put'
            })
            .done(function(data) {
                                  $('#compte,  #deconnexion, #admin').addClass('hidden');
              $('#connexion').removeClass('hidden');
            });
            return false;
          });
        });
      </script>
    @endif
    @yield('scripts')
  </body>
</html>
```

J’ai utilisé Bootstrap 3 avec une barre de navigation (sans le responsive pour alléger le code). Les items du menu s’adaptent selon l’utilisateur connecté et ses permissions.

Il y a un template spécifique pour les formulaires :

```
@extends('Shared.views.main')

@section('contenu')
        <div class="jumbotron col-sm-offset-3 col-sm-6">
                <h2>
                        @yield('titre')
                </h2>
                <br>
                @if($message = Session::get('success'))
                         <div class="alert alert-success alert-
```

```
dismissable">
                                        <button type="button"
class="close" data-dismiss="alert">×</button>
                                {{ $message }}
                        </div>
                @endif
                @if($message = Session::get('error'))
                          <div class="alert alert-danger alert-
dismissable">
                                        <button type="button"
class="close" data-dismiss="alert">×</button>
                                {{ $message }}
                        </div>
                @endif
                @if($message = Session::get('info'))
                           <div class="alert alert-info alert-
dismissable">
                                        <button type="button"
class="close" data-dismiss="alert">×</button>
                                {{ $message }}
                        </div>
                @endif
                @yield('form')
        </div>
@stop
```

On prévoit une mise en page et l'affichage éventuel de messages.

Les langues

Pour la partie frontend j'ai prévu une possibilité d'avoir plusieurs langages. Les fichiers avec les textes se trouvent dans le dossier lang/fr :

Il s'agit de simples tableaux de type clé-valeur.

Le throttling

Ce terme difficile à traduire correspond à une sécurité prévue dans Sentry. Voyons de quoi il s'agit. Lorsque des tentatives de connexions infructueuses ont lieu Sentry les mémorise dans la table throttle. Il y a 3 types prévus :

- **Global** : (toutes les tentatives d'où qu'elles viennent et quel que soit le compte)
- **IP** : tentatives issues d'une adresse IP spécifique,
- **User** : tentative sur un compte spécifique.

Si vous regardez dans le fichier app/config/packages/cartalyst/sentry/config.php :

Vous trouvez la configuration de ces protections :

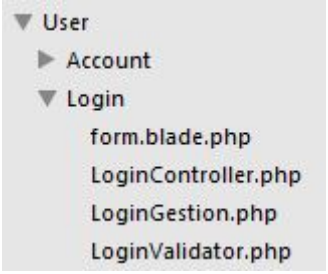
```
'throttling' => array(

        'model' => 'Cartalyst\Sentry\Throttling\EloquentThrottle',

        'global' => array(

                'interval' => 900,

                'thresholds' => array(
                        10 => 1,
                        20 => 2,
                        30 => 4,
                        50 => 8,
                        50 => 16,
                        60 => 12
                ),

        ),
```

```
        'ip' => array(
                'interval' => 900,
                'thresholds' => 5,
        ),
        'user' => array(
                'interval' => 900,
                'thresholds' => 5,
        ),
),
```

D’une part vous pouvez paramétrer la durée de blocage de l’accès, par défaut on a 900 secondes. D’autre part vous déterminez le nombre de tentatives d’accès. Par défaut on a 5 pour les IP et user. Pour le global vous avez un tableau dans lequel la clé est le nombre d’accès et la valeur du délai).

La connexion

Commençons par voir la connexion. On a 4 fichiers concernés :



Commençons par voir le contrôleur. La route login avec le verbe get conduit sur la méthode getIndex :

```
public function getIndex()
{
        return View::make('User.Login.form');
}
```

Ce qui a pour effet de créer la vue de la connexion avec le fichier form.blade.php :

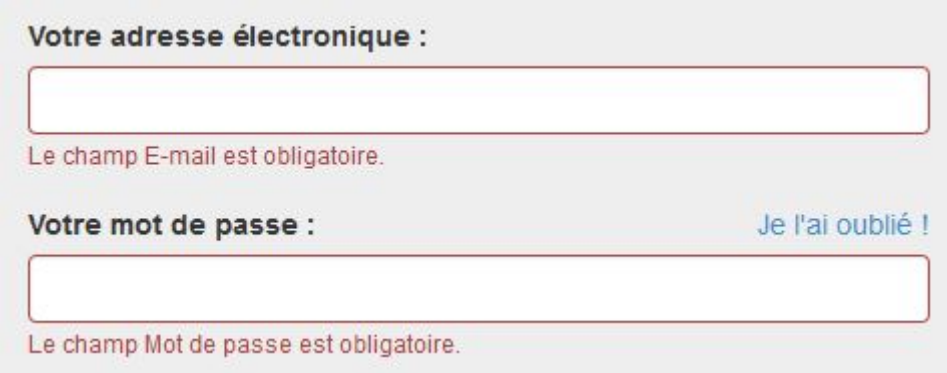
```
@extends('Shared.views.form')

@section('titre')
        {{ trans('user.connectez_vous') }}
@stop

@section('form')
        {{ Form::open(array('url' => 'login')) }}
                {{ Former::control('email', $errors, 'email', trans('user.email')) }}
                    {{ Former::pass($errors, 'password', trans('user.passe'), true) }}
                            {{  Former::check('souvenir', trans('user.se_rappeler')) }}
                {{ Former::button_submit(trans('user.envoyer')) }}
        {{ Form::close() }}
        <br>
          {{ link_to('register', trans('user.creer_compte'), array('class' => 'btn btn-warning')) }}
@stop
```

Avec cette apparence :

Le formulaire est traité par la méthode postIndex du contrôleur :

```
public function postIndex()
{
        if ($this->validation->fails()) {
                return Redirect::to('login')
                ->withInput()
                ->withErrors($this->validation->errors());
        }

        $result = $this->gestion->authenticate();

        if($result === true) {
                return Redirect::intended();
        }

         return Redirect::to('login')->withInput()->with('error',
$result);
}
```

En premier on valide avec renvoi du formulaire en cas d’erreur. Par exemple :



Puis on tente l'authentification en appelant la méthode authenticate prévue dans la classe de gestion :

```
public function authenticate()
{
        try
        {
                    if(Sentry::authenticate(Input::all(),
Input::get('souvenir'), true)){
                        return true;
                }
                return trans('user.mauvais_passe');
        }
        catch (NotActivatedException $e)
        {
                return trans('user.compte_pas_actif');
        }
        catch (ThrottlingException $e)
        {
                $time = $e->getDelay();
                 return trans('user.throttling_' . $e->getType(),
compact('time'));
        }
}
```

On utilise la méthode de même nom de Sentry en transmettant les entrées, la valeur de la case à cocher pour la mémorisation de l'utilisateur. Les trois erreurs possibles sont :

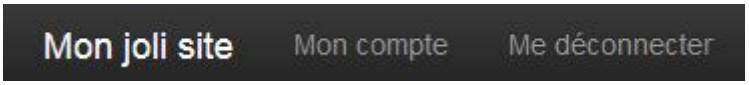
- un mauvais mot de passe (on ne peut pas avoir une mauvaise adresse parce qu'on vérifie au niveau de la validation que l'adresse saisie existe) :

- l’utilisateur n’est pas activé :

- il y a un blocage pour des raisons de sécurité :

Ici j’ai pris le cas du throttling pour l’utilisateur, on pourrait avoir les deux autres possibilités

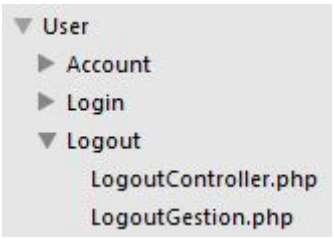
Lorsque l’utilisateur est connecté le menu s’adapte :



Là on a le cas de l'utilisateur de base, pour un administrateur on a en plus l'accès à l'administration :

La déconnexion

Le symétrique de la connexion est la déconnexion. Le traitement est plus simple, on a que deux classes :



Le contrôleur attend une requête Ajax avec le verbe PUT :

```
public function putIndex()
{
        if(Request::ajax())
        {
                $this->gestion->logout();
                return Response::make('', 200);
        }
}
```

La classe de gestion fait appel à Sentry pour déconnecter l'utilisateur :

```
public function logout()
{
        Sentry::logout();
}
```

Le code Javascript pour l'envoi de la requête est inséré dans la vue générale lorsque on a un utilisateur connecté :

```
@if(\Sentry::check())
  <script>
    $(function(){
      $('#deconnexion a').click(function(e){
```

```
        e.preventDefault();
        $.ajax({
          url: 'logout',
          type: 'put'
        })
        .done(function(data) {
          $('#compte, #deconnexion, #admin').addClass('hidden');
          $('#connexion').removeClass('hidden');
        });
        return false;
      });
    });
  </script>
@endif
```

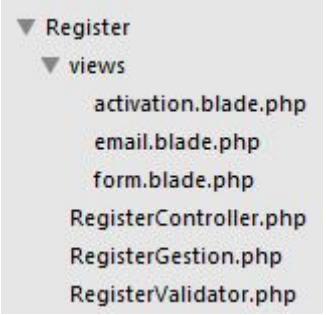
Lorsqu'on reçoit la réponse on adapte le menu en jouant avec la classe hidden de Bootstrap. Le fait d'utiliser Ajax permet de garder intact le contexte dans lequel l'utilisateur se trouve sur le site.

Enregistrement et activation

Un utilisateur doit pouvoir s'enregistrer sur le site. Cela s'effectue en deux étapes :

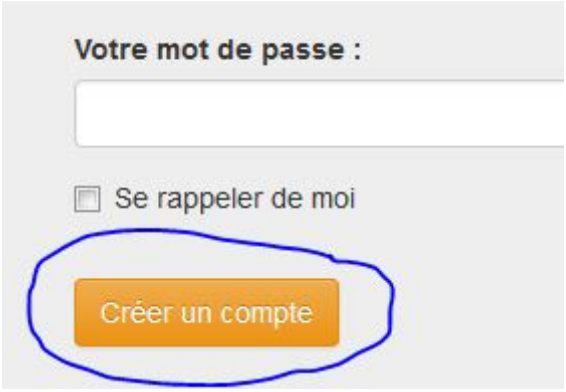
- l'enregistrement avec saisi des informations : nom, Email, mot de passe,
- l'activation pour s'assurer que l'Email est valide.

Tout ce qui concerne cette partie se situe dans le dossier register :



Enregistrement

Pour s'enregistrer il faut cliquer sur le bouton correspondant dans le formulaire de connexion :



Dans le contrôleur RegisterController on tombe sur la méthode getIndex :

```
public function getIndex()
{
        return View::make('User.Register.views.form');
}
```

Ça nous renvoie sur la vue form.blade.php :

```
@extends('Shared.views.form')

@section('titre')
        {{ trans('user.enregistrez_vous') }}
@stop

@section('form')
        {{ Form::open(array('url' => 'register')) }}
                {{ Former::control('text', $errors, 'first_name',
trans('user.prenom')) }}
                 {{ Former::control('text', $errors, 'last_name',
trans('user.nom')) }}
                  {{ Former::control('email', $errors, 'email',
trans('user.email')) }}
                     {{ Former::pass($errors, 'password',
trans('user.passe')) }}
                {{ Former::pass($errors, 'password_confirmation',
trans('user.passe_confirmation')) }}
               {{ Former::button_submit(trans('user.envoyer')) }}
```

```
        {{ Form::close() }}
@stop
```

Avec cet aspect :

Le formulaire est traité par la méthode postIndex du contrôleur RegisterController :

```
public function postIndex()
{
        if ($this->validation->fails()) {
          return Redirect::to('register')
          ->withInput()
          ->withErrors($this->validation->errors());
        }

        if($this->gestion->register()) {
```

```
                return Redirect::to('login')->with('info',
trans('user.email_envoye'));
        }
```

Un fois passée la validation on appelle la méthode register de la classe RegisterGestion :

```
public function register()
{
          $user = Sentry::register(array_merge(Input::all(),
['permissions' => ['user' => true]]));

            $activation = Activation::exists($user) ?:
Activation::create($user);

        $code = $activation->code;
        $id = $user->id;

  return Mail::send('user.register.views.email', compact('code',
'id'), function($message) use ($user)
        {
$message->to($user->email)->subject(trans('user.activez_compte'));
        });
}
```

L'utilisateur est enregistré avec la méthode register de Sentry.

Activation

Dans la même méthode on crée une activation, on récupère le code et on envoie un Email pour l'activation. L'Email contient un lien qui envoie à la méthode getActivation du contrôleur :

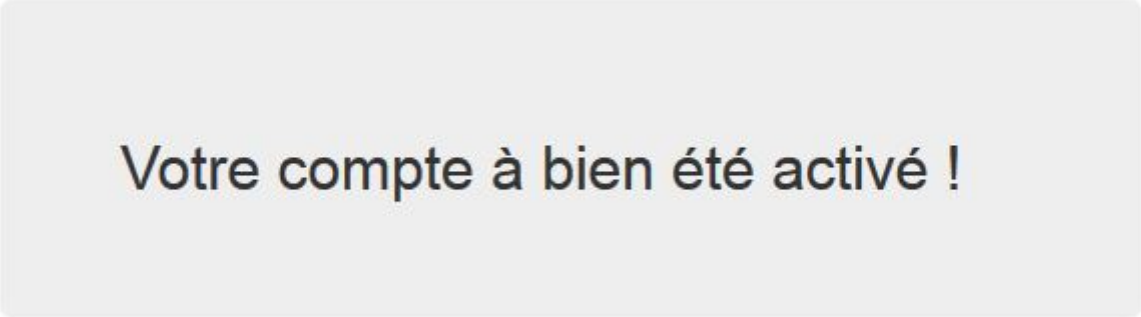
```
public function getActivation($code, $id)
{
        if(ctype_digit($id))
        {
                $info = $this->gestion->activation($code, $id);
                                                        return
View::make('user.register.views.activation',  array('info'  =>
$info));
        }
```

```
        App::abort(404);
}
```

C’est la méthode activation de la gestion qui effectue le traitement :

```
public function activation($code, $id)
{
        if($user = Sentry::findById($id))
        {
                if (Activation::complete($user, $code))
                {
                        return trans('user.compte_active');
                }
                return trans('user.compte_pas_active');
        }
        return trans('user.pas_dans_base');
}
```

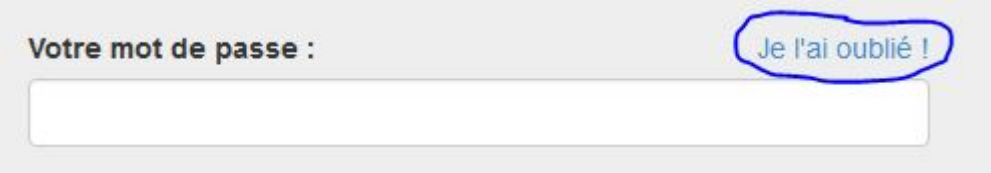
L’utilisateur est trouvé par son identifiant et la méthode findById de Sentry. L’activation est réalisée par la méthode complete de la classe Activation. Si tout se passe bien on en informe l’utilisateur :



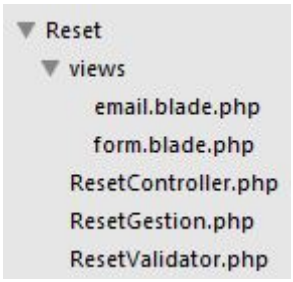
Oubli du mot de passe

La demande

Pour les étourdis il est prévu la possibilité d’enregistrer un nouveau mot de passe. On commence par cliquer sur le lien prévu au niveau du formulaire de connexion :



Tout le code qui concerne cette partie se situe dans le dossier reset :



Le clic envoie sur la méthode getIndex du contrôleur qui appelle la vue form.blade.php :

```
@extends('Shared.views.form')

@section('titre')
        {{ trans('user.reinitialisation_passe') }}
@stop

@section('form')
        {{ Form::open(array('url' => 'reset')) }}
                {{ Former::control('email', $errors, 'email',
trans('user.email')) }}
                {{ Former::button_submit('Envoyer !') }}
        {{ Form::close() }}
@stop
```

Avec cet aspect :

Pour identifier l'utilisateur on lui demande son adresse Email. Le formulaire est traité par la méthode postIndex du contrôleur :

```
public function postIndex()
{
        if ($this->validation->fails()) {
          return Redirect::to('reset')
          ->withInput()
          ->withErrors($this->validation->errors());
        }

        if($this->gestion->reset()) {
                return Redirect::to('reset')
                ->withInput()
                ->with('info', trans('user.email_envoye'));
        }

            return Redirect::to('register')->with('error', trans('user.email_pas_envoye'));
}
```

La partie intéressante réside dans l'appel de la méthode reset de la gestion :

```
public function reset()
{
        $email = Input::get('email');

        $user = Sentry::findByCredentials(compact('email'));
```

```
                $reminder = Reminder::exists($user) ?:
Reminder::create($user);

        $code = $reminder->code;
        $id = $user->id;

            return Mail::send('User.Reset.views.email',
compact('code', 'id'), function($message) use ($email)
        {
$message->to($email)->subject(trans('user.reinitialisation_passe')
);
        });
}
```

On utilise la méthode findByCredentials de Sentry pour retrouver l'utilisateur qui correspond à l'Email. On vérifie au passage si il n'a pas déjà été effectué une demande et on crée un reminder avec la classe Reminder et sa méthode create.

Supposons que c'est l'utilisateur de base qui a oublié son mot de passe. Quand il envoie le formulaire il reçoit une notification :

Si on regarde dans la table reminders on trouve un enregistrement :

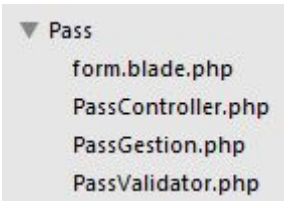
id	user_id	code	completed
1	3	bLTf2yH3v8T815REUNXS6V80GROvHfd1	0

On envoie dans le lien de l'Email le code et l'id pour retrouver

l’utilisateur au retour.

Le nouveau mot de passe

Le traitement pour le nouveau mot de passe se situe dans le dossier pass :



Le lien dans l’Email envoie sur la méthode getIndex du contrôleur :

```
public function getIndex($code, $id)
{
          return View::make('User.Pass.form', compact('code', 'id'));
}
```

On récupère le code et l’id dans les paramètres. On appelle la vue form.blade.php :

```
@extends('shared.views.form')

@section('titre')
        {{ trans('user.votre_nouveau_passe') }}
@stop

@section('form')
        {{ Form::open(array('url' => 'pass/index/' . $code . '/' . $id)) }}
                  {{ Former::control('email', $errors, 'email', trans('user.email')) }}
                     {{ Former::pass($errors, 'password', trans('user.passe')) }}
                 {{ Former::pass($errors, 'password_confirmation', trans('user.passe_confirmation')) }}
                {{ Former::button_submit(trans('user.envoyer')) }}
        {{ Form::close() }}
@stop
```

Avec cet aspect :

Le traitement du formulaire est effectué par la méthode postIndex du contrôleur :

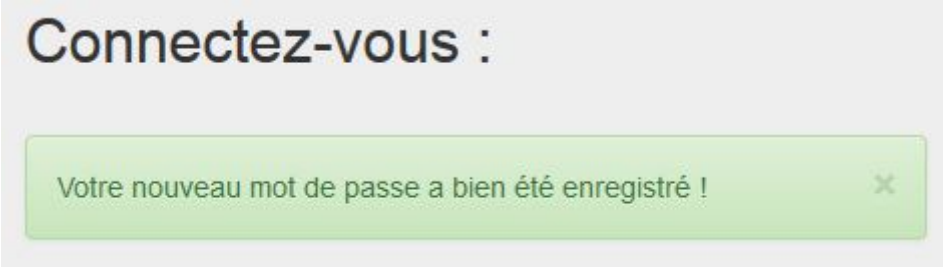
```
public function postIndex($code, $id)
{
        if ($this->validation->fails()) {
          return Redirect::back()
          ->withInput()
          ->withErrors($this->validation->errors());
        }

                if($this->gestion->complete($code, $id,
Input::get('password')) === true) {
                return Redirect::to('login')
                ->with('success', trans('user.passe_enregistre'));
        }

        return Redirect::back()
        ->with('error', trans('user.passe_pas_enregistre'));
}
```

Le contrôleur demande à la méthode complete de la gestion de

réaliser le traitement avec Sentry :

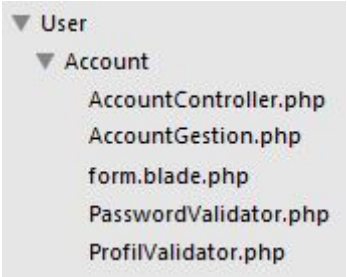
```
public function complete($code, $id, $password)
{
        if($user = Sentry::findById($id))
        {
                if (Reminder::complete($user, $code, $password))
                {
                        return true;
                }
                                                        return
trans('user.reinitialisation_pas_correcte');
        }
        return trans('user.pas_dans_base');
}
```

C’est la classe Reminder qui est chargée du traitement avec sa méthode complete. Si tout se passe bien on notifie en renvoyant au formulaire de connexion :



Gestion du compte

L’utilisateur peut gérer ses informations. Le code se trouve dans le dossier account :



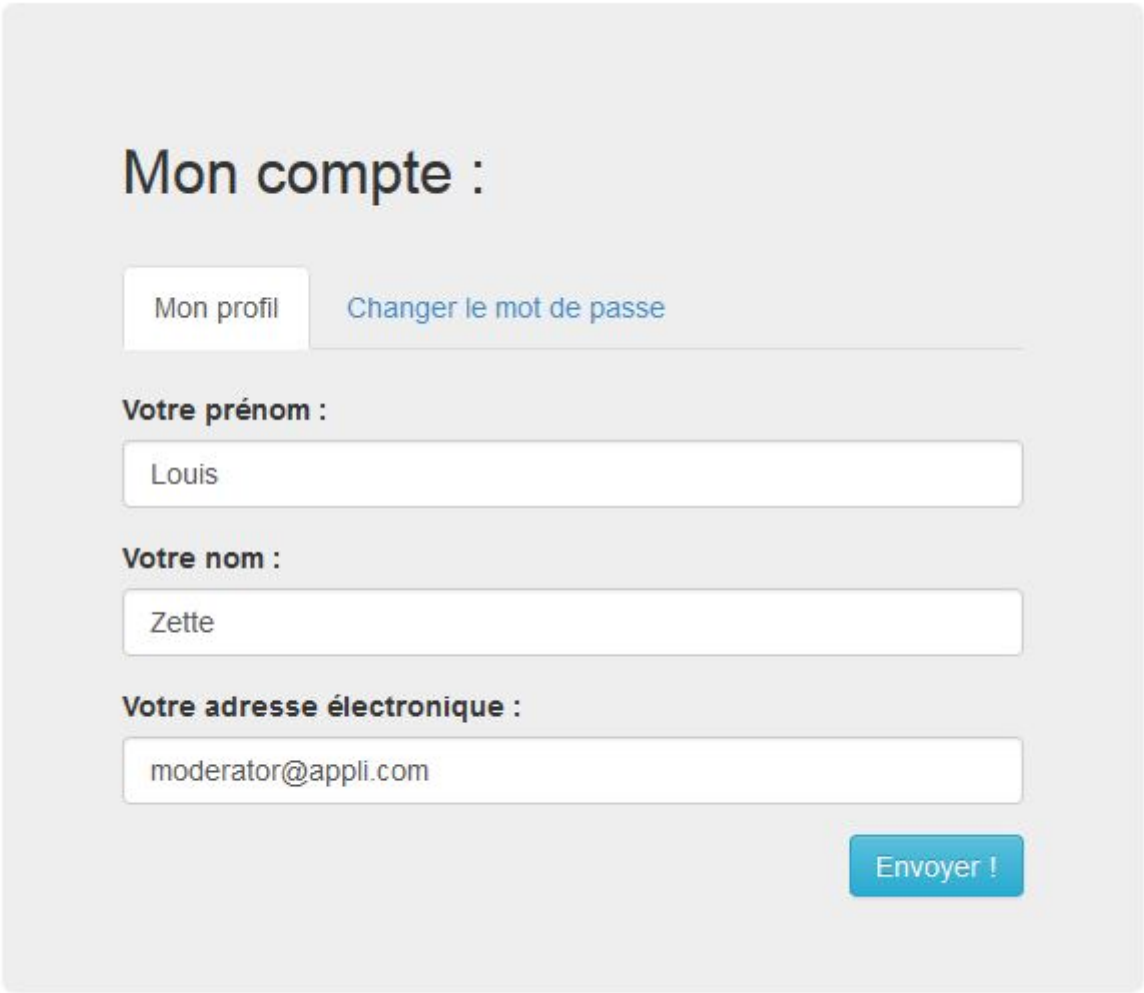
On trouve un double formulaire dans la vue form.blade.php :

```
@extends('Shared.views.form')
```

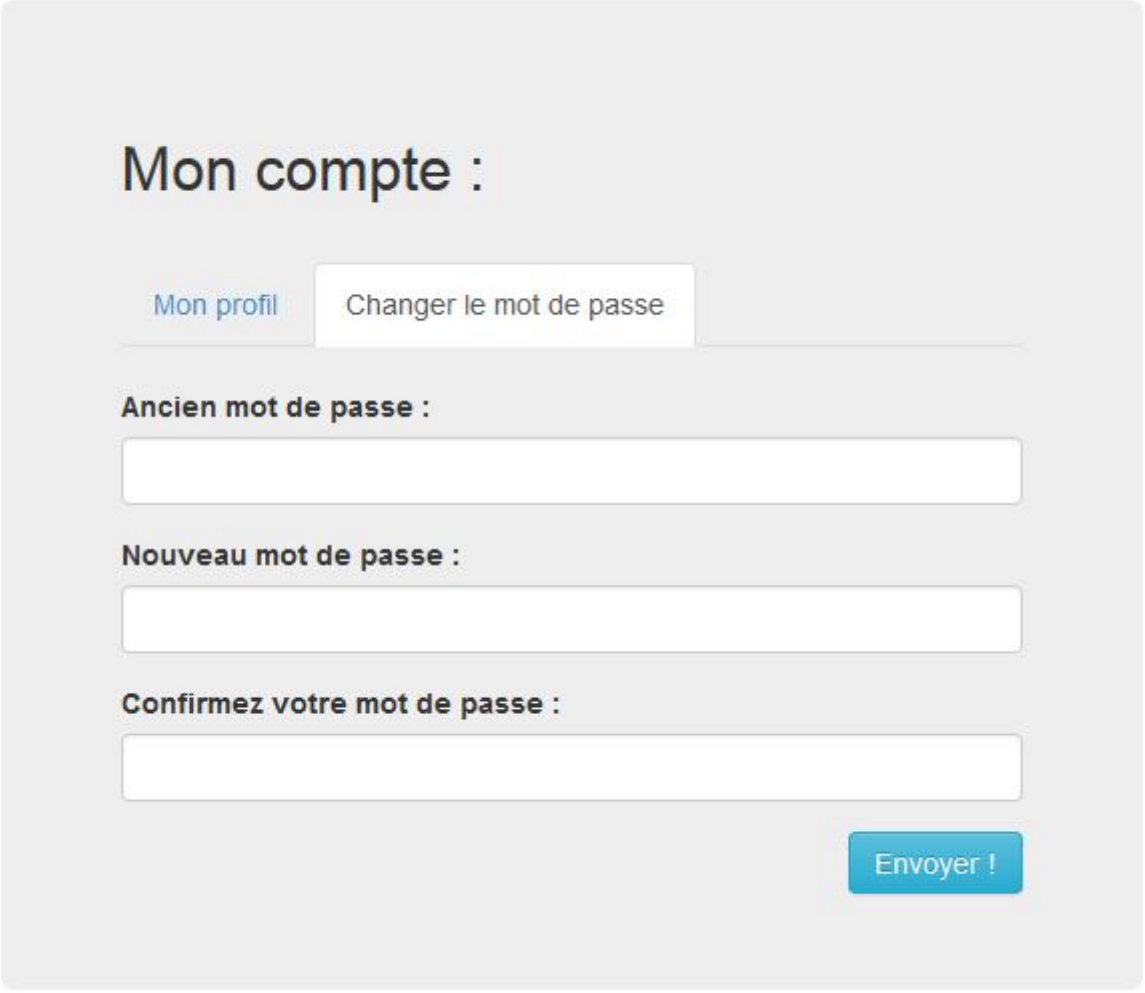
```
@section('titre')
        {{ trans('user.mon_compte') }}
@stop

@section('form')
        <ul class="nav nav-tabs">
          <li {{ Session::has('pass')? '' : 'class="active"' }}><a
href="#profil"  data-toggle="tab">{{  trans('user.mon_profil')
}}</a></li>
          <li {{ Session::has('pass')? 'class="active"' : '' }}><a
href="#passe"  data-toggle="tab">{{  trans('user.changer_passe')
}}</a></li>
        </ul>
        <div class="tab-content">
           <div class="tab-pane {{ Session::has('pass')? '' :
'active' }}" id="profil">
                <br>
                {{ Form::open(array('url' => 'account/profil/' .
$user->id)) }}
                                    {{ Former::control('text',
$errors, 'first_name', trans('user.prenom'), $user->first_name) }}
                                    {{ Former::control('text',
$errors, 'last_name', trans('user.nom'), $user->last_name) }}
                                   {{ Former::control('email',
$errors, 'email', trans('user.email'), $user->email) }}
                                                                {{
Former::button_submit(trans('user.envoyer')) }}
                        {{ Form::close() }}
          </div>
          <div class="tab-pane {{ Session::has('pass')? 'active' :
'' }}" id="passe">
                <br>
                                  {{ Form::open(array('url' =>
'account/password')) }}
                                     {{ Former::pass($errors,
'old_password', trans('user.vieux_passe')) }}
                                     {{ Former::pass($errors,
'password', trans('user.nouveau_passe')) }}
                                     {{ Former::pass($errors,
'password_confirmation', trans('user.passe_confirmation')) }}
                                                                {{
Former::button_submit(trans('user.envoyer')) }}
                        {{ Form::close() }}
```

```
          </div>
        </div>
@stop
```

Avec cet aspect :



Le premier formulaire permet de changer nom, prénom et adresse Email. Le second permet de changer son mot de passe :



Je ne vais pas exposer tout le code qui ne concerne pas directement Sentry. La classe de gestion comprend deux méthodes :

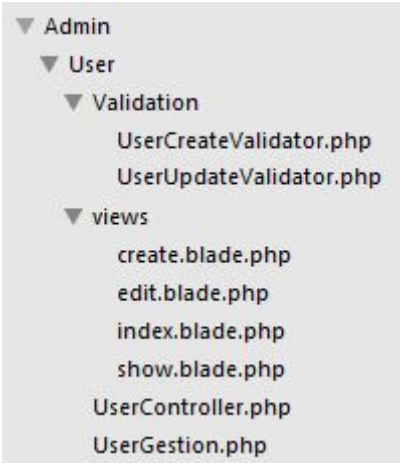
```
public function update()
{
        return Sentry::update($this->getUser(), Input::all());
}

public function checkPassword()
{
        return Sentry::stateless(
                array(
                        'email' => $this->getUser()->email,
                        'password' => Input::get('old_password')
                )
        );
}
```

La première (update) est chargée de la mise à jour des informations de l'utilisateur. On voit qu'on fait appel à la méthode éponyme de Sentry.

La seconde (checkPassword) fait appel à la méthode stateless de Sentry qui permet d'effectuer une connexion factice pour vérifier que l'ancien mot de passe saisi par l'utilisateur est le bon.

L'administration

Si on se connecte avec l'Email admin@appli.com on peut accéder à l'administration. Le code se trouve dans le dossier admin/user :



Le filtre

On a vu que dans les routes on utilise le filtre access pour accéder à cette partie :

```
Route::group(array('before' => 'access:admin'), function()
{
        Route::resource('user', 'Lib\admin\User\UserController');
});
```

On trouve ce filtre dans le fichier app/filters.php :

```
Route::filter('access', function($route, $request, $right)
{
        if(Sentry::check()) {
                if(!Sentry::getUser()->hasAccess($right)) {
                        return Redirect::to('/');
                }
        } else {
                return Redirect::to('login');
        }
});
```

J’ai prévu un paramètre pour rendre ce filtre adaptable. On utilise la méthode hasAccess de Sentry pour tester les droits de l’utilisateur connecté. Dans notre cas on demande si c’est un administrateur (admin).

La liste des utilisateurs

Comme les utilisateurs sont gérés directement par Sentry on va faire appel à ses méthodes. Toutefois on ne pourra pas tout faire, comme par exemple afficher la liste des utilisateurs. On va devoir référencer dans la classe de gestion (UserGestion) le modèle de Sentry :

```
use Cartalyst\Sentry\Users\EloquentUser as User;
```

Il devient comme ça facile d’envoyer une liste paginée :

```
public function index($nombre)
{
        $users = User::paginate($nombre);
        return compact('users');
}
```

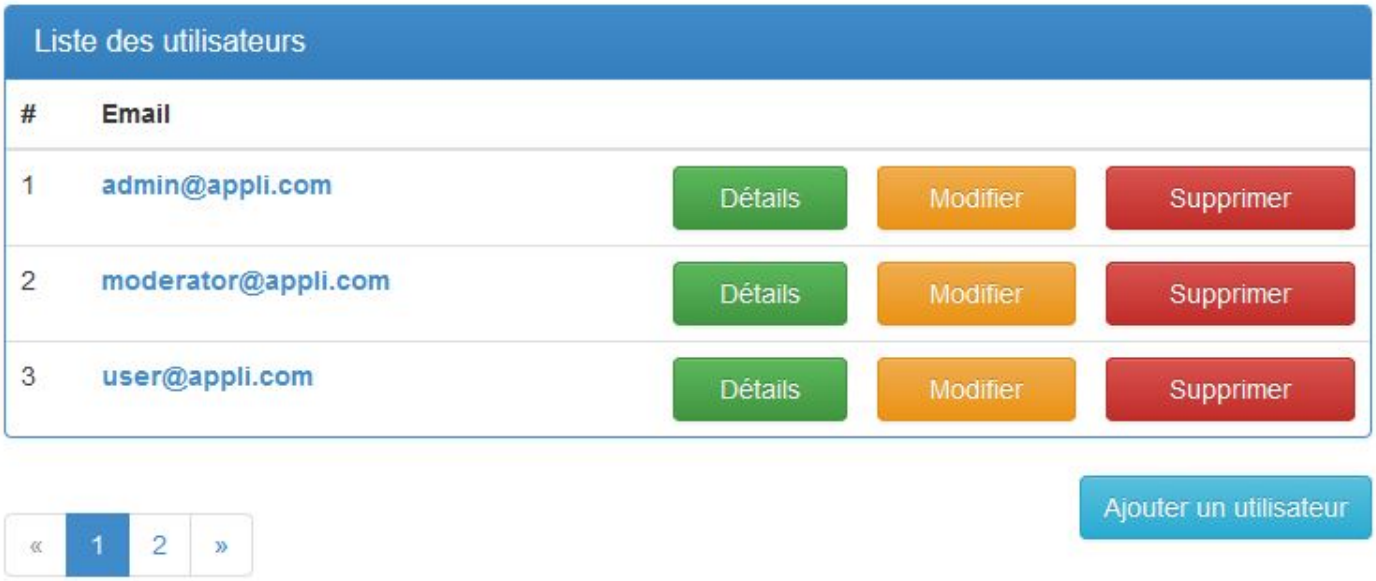
Cette liste est envoyée dans la vue index.blade.php :

```
@extends('Shared.views.main')

@section('contenu')
        <br>
        <div class="col-sm-offset-2 col-sm-8">
                @if(Session::has('success'))
                        <div class="alert alert-success alert-
dismissable">
                                        <button type="button"
class="close" data-dismiss="alert">×</button>
                                {{ Session::get('success') }}
                        </div>
                @endif
                <div class="panel panel-primary">
                        <div class="panel-heading">
                                <h3 class="panel-title">Liste des
utilisateurs</h3>
                        </div>
```

```
                        <table class="table">
                                <thead>
                                        <tr>
                                                <th>#</th>
                                                <th>Email</th>
                                                <th></th>
                                                <th></th>
                                                <th></th>
                                        </tr>
                                </thead>
                                <tbody>
                                  @foreach ($users as $user)
                                          <tr>
                                              <td>{{ $user->id }}</td>
                                              <td class="text-primary"><strong>{{ $user->email }}</strong></td>
                                                                  <td>{{ link_to_route('user.show',   'Détails',   array($user->id), array('class' => 'btn btn-success btn-block')) }}</td>
                                                                  <td>{{ link_to_route('user.edit',   'Modifier',   array($user->id), array('class' => 'btn btn-warning btn-block')) }}</td>
                                                                      <td><button type="button" class="btn btn-danger btn-block" id="{{ $user->id }}">Supprimer</button></td>
                                          </tr>
                                  @endforeach
                          </tbody>
                        </table>
                </div>
                  {{ link_to_route('user.create', 'Ajouter un utilisateur', null, array('class' => 'btn btn-info pull-right')) }}
                {{ $users->links(); }}
        </div>
@stop

@section('scripts')
  <script>
    $(function(){
      $('.btn-danger').click(function(e) {
        if(confirm('Vraiment supprimer cet utilisateur ?')) {
```

```
                e.preventDefault();
                $.ajax({
                  url: 'user/' + $(this).attr('id'),
                  type: 'delete'
                })
                .done(function(id) {
                  $('#' + id).parents('tr').remove();
                });
                return false;
              }
      });
    });
  </script>
@stop
```

Avec cet aspect :



Pour faire apparaître la pagination j’ai ajouté des utilisateurs.

Fiche d’un utilisateur

Si on clique sur le bouton « Détails » on obtient la fiche de l’utilisateur avec ses principales informations :

On trouve le traitement dans la méthode show de la gestion (UserGestion) :

```
public function show($id)
{
        $user = Sentry::findById($id);
        $permission = head(array_keys($user->permissions));
        return compact('user', 'permission');
}
```

On utilise la méthode findById de Sentry pour tout récupérer. Pour la permission on extrait la clé de tableau des permissions. La méthode head fait partie des helpers de Laravel pour les tableaux, elle permet de récupérer le premier élément.

La vue (show.blade.php) est classique :

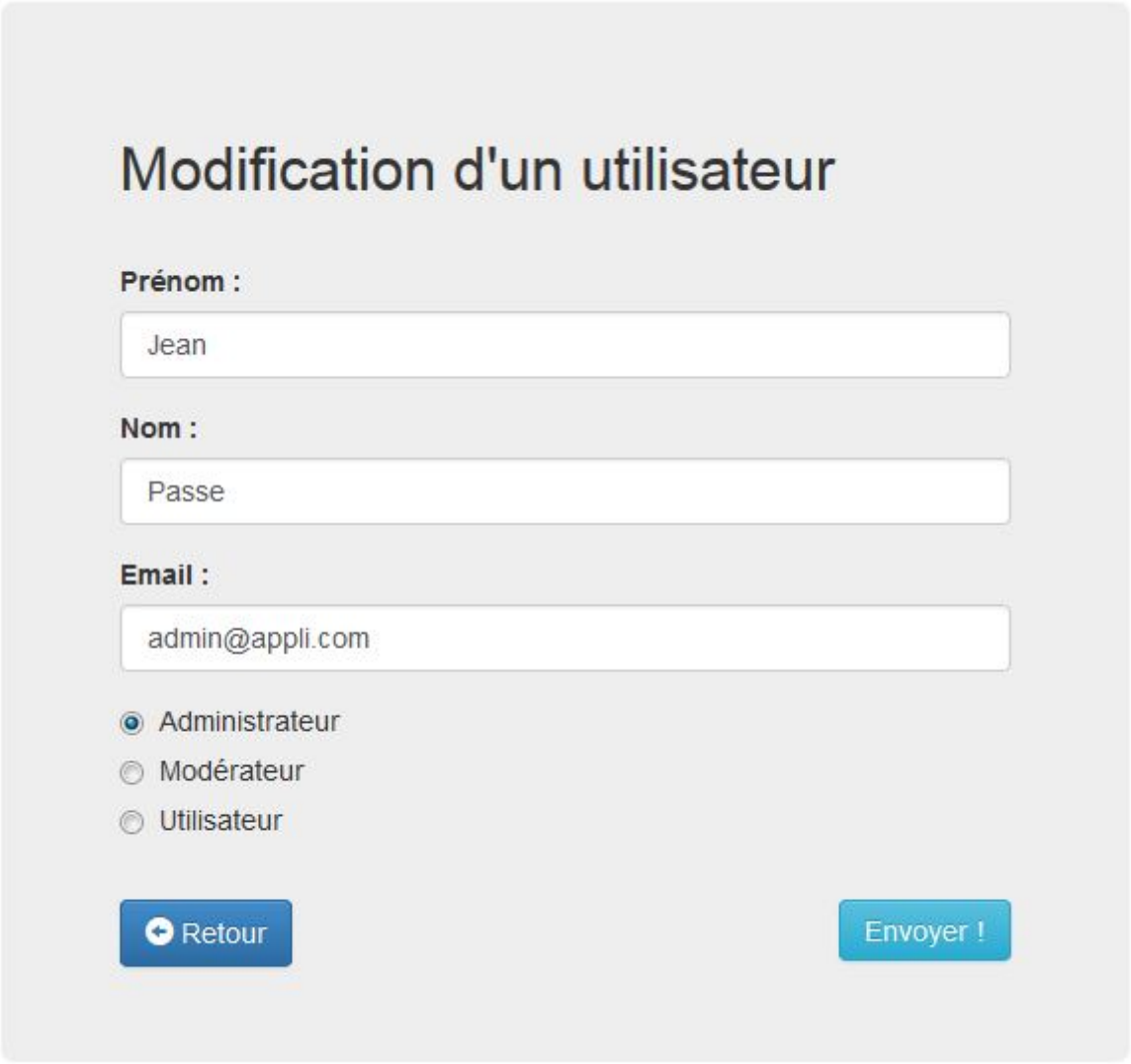
```
@extends('Shared.views.main')

@section('contenu')
        <div class="row col-md-offset-3 col-md-6">
                <div class="panel panel-primary">
                        <div class="panel-heading">Détail d'un
utilisateur</div>
                    <div class="panel-body">
                            <p>Prénom : {{{ $user->first_name
}}}</p>
                             <p>Nom : {{{ $user->last_name
}}}</p>
```

```
                                    <p>Email : {{{ $user->email }}}</p>
                                    <p>Permission : {{ $permission }}</p>
                                    <p>Date d'inscription : {{ \Carbon\Carbon::createFromFormat('Y-m-d H:i:s', $user->created_at)->format('j/m/Y') }}</p>
                                    @if($user->last_login !== null)
                                        <p>Dernière connexion : {{ \Carbon\Carbon::createFromFormat('Y-m-d H:i:s', $user->last_login)->format('j/m/Y') }}</p>
                                    @endif
                                </div>
                            </div>
                            <a href="javascript:history.back()" class="btn btn-primary">
                                <span class="glyphicon glyphicon-circle-arrow-left"></span> Retour
                            </a>
                        </div>
@stop
```

Remarquez l'utilisation de Carbon (qui est une dépendance de Laravel) pour gérer les dates avec simplicité.

Modifier un utilisateur

Si on clique sur le bouton « Modifier » on obtient un formulaire :



On peut modifier : le prénom, le nom, l'Email et la permission.

La mise en oeuvre des modifications se fait dans la méthode update de la gestion :

```
public function update($id)
{
            return  Sentry::update(Sentry::findById($id),
$this->getInput());
}
```

On se contente d'utiliser la méthode update de Sentry.

Le reste du code ne concerne pas directement Sentry et je ne le commente pas.

Supprimer un utilisateur

Si on cliquer sur le bouton « Supprimer » on peut supprimer un utilisateur. Par sécurité j'ai prévu une petite fenêtre gérée en Javascript pour la confirmation.

La suppression s'effectue dans la méthode destroy de la gestion :

```
public function destroy($id)
{
  Sentry::findById($id)->delete();
}
```

Là encore on utilise une méthode de Sentry (delete).

J'ai jugé préférable de gérer cette action avec Ajax :

```
$(function(){
        $('.btn-danger').click(function(e) {
                  if(confirm('Vraiment supprimer cet utilisateur
?')) {
                        e.preventDefault();
                        $.ajax({
                                url: 'user/' + $(this).attr('id'),
                                type: 'delete'
                        })
                        .done(function(id) {
                                                        $('#'  +
id).parents('tr').remove();
                        });
                        return false;
                }
        });
});
```

On envoie la requête avec le verbe DELETE. Au retour on efface la ligne au niveau du DOM. Dans le contrôleur on vérifie qu'on reçoit bien une requête de type Ajax :

```
public function destroy($id)
{
        if(Request::ajax())
        {
                $this->gestion->destroy($id);
```

```
                return Response::make($id, 200);
        }
}
```

On retourne l'identifiant de l'utilisateur pour effacer la bonne ligne .

Créer un utilisateur

Si on clique sur le bouton « Ajouter un utilisateur » on obtient ce formulaire :

Il ressemble évidemment beaucoup à celui chargé des modifications puisque nous avons les mêmes contrôles.

Au niveau de la gestion la méthode store effectue la création effective :

```
public function store()
```

```
{
        Sentry::registerAndActivate(array_merge($this->getInput(),
['password' => 'password']));
}
```

Ici on utilise la méthode registerAndActivate de Sentry. En effet, comme l’utilisateur est directement créé par l’administrateur on va éviter l’étape de l’activation.

Le reste du code n’appelle pas de commentaire particulier.