Envoyer des mails dans un projet Symfony

Objectifs pédagogiques:

A l'issue de cette séance, vous serez capables de construire et d'envoyer des mails en utilisant la bibliothèque Mailer de Symfony.

Cheminement

Installation de Mailer

Pour pouvoir utiliser la bibliothèque Mailer dans un projet Symfony, il faut l'installer en exécutant la commande suivante :

```
composer require symfony/mailer
```

Attention: Si vous avez créé votre projet Symfony avec "webapp" (c'est le cas si vous avez suivi les cours précédents), cette commande n'est pas nécessaire! La librairie est déjà installée dans votre projet. Vous pouvez vérifier dans votre fichier composer.json la présence de la ligne suivante :

```
{
    //...
    "require": {

        //...
        "symfony/mailer": "6.1.*",
```

Configuration de la couche transport dans le fichier .env.local

Une fois le Mailer installé, il vous faut configurer le serveur de mail qui sera utilisé. Pour ce faire, allez dans le fichier .env.local et modifiez la ligne suivante :

```
MAILER_DSN=smtp://votre_identifiant:votre_mot_de_passe@smtp.example.com:port
```

Notes Si vous avez installé MailHog, vous pouvez modifier la ligne ci-dessus comme suit :

```
MAILER_DSN=smtp://localhost:1025
```

> Comme MailHog, il existe d'autres outils comme Mailtrap qui peuvent simuler un SMTP et récupérer l'ensemble des mails envoyés par Symfony

Créer et envoyer un message avec Mailer

Si vous avez déjà utilisé la librairie PHPMailer, vous trouverez beaucoup de similitudes entre le Mailer de Symfony et cette librairie. En effet, si l'on regarde l'exemple fourni dans la documentation de Symfony, on remarque bon nombre de méthodes présentes aussi dans PHPMailer (dont send()):

```
// src/Controller/MailerController.php
namespace App¥Controller;

use Symfony¥Bundle¥FrameworkBundle¥Controller¥AbstractController;
use Symfony¥Component¥HttpFoundation¥Response;
```

```
use Symfony¥Component¥Mailer¥MailerInterface;
use Symfony¥Component¥Mime¥Email;
use Symfony¥Component¥Routing¥Annotation¥Route;

class MailerController extends AbstractController
{
    #[Route('/email')]
    public function sendEmail(MailerInterface $mailer): Response
    {
        $email = (new Email())
            ->from('hello@example.com')
            ->to('you@example.com')
            //->cc('cc@example.com')
            //->bcc('bcc@example.com')
            //->replyTo('fabien@example.com')
            //->priority(Email::PRIORITY_HIGH)
            ->subject('Time for Symfony Mailer!')
            ->text('Sending emails is fun again!')
            ->html('<p>See Twig integration for better HTML integration!</p>');

        $mailer->send($email);

        // ...
    }
}
```

Explications :

- Dans l'exemple ci-dessus, nous avons une route (`#[Route('/email')]`), associée à une fonction (`sendEmail()`) dans le contrôleur `MailerController`.
- L'interface de Mailer est injectée dans la fonction `public function sendEmail(MailerInterface $mailer)`
- Pour utiliser cette interface, tout comme la classe `Email`, il faut faire ce que l'on appelle du `type-hinting`, c'est-à-dire préciser le fait que les variable(s) passées en tant que paramètres doivent être du même type que ces classes (elles sont dans les "use" en haut du fichier - lignes 51 et 52)
- Dans la construction de l'email il y a l'expéditeur (`from`), le destinataire (`to`), le sujet (`subject`), le corps du mail
- Le message est envoyé èn même temps sous format texte brut (`text`) et html (`html`)

Même si fonctionnelle, cette manière de générer un mail en format `html` n'est pas idéale, surtout quand veut envoyer des variables par exemples. Le `Mailer` propose une classe appelée `TemplatedEmail` qui permet de construire un email plus élaboré et plus riche en fonctionnalités.

Avant de l'utiliser, assurez-vous d'avoir installé le bundle Twig. Exécutez la commande suivante:

```
composer require symfony/twig-bundle
```

Reprenons l'exemple de la documentation de `Symfony` et modifions quelques lignes dans le code :

```
// src/Controller/MailerController.php
namespace App¥Controller;
```

```
use Symfony¥Bundle¥FrameworkBundle¥Controller¥AbstractController;
use Symfony¥Component¥HttpFoundation¥Response;
use Symfony¥Component¥Mailer¥MailerInterface;
//use Symfony¥Component¥Mime¥Email;
use Symfony¥Bridge¥Twig¥Mime¥TemplatedEmail;
use Symfony¥Component¥Mime¥Address;
use Symfony¥Component¥Routing¥Annotation¥Route;

class MailerController extends AbstractController
{
    #[Route('/email')]
    public function sendEmail(MailerInterface $mailer): Response
    {
        $email = (new TemplatedEmail())
            ->from('hello@example.com')
//            ->to('you@example.com')
            ->to(new Address('ryan@example.com'))
            //->cc('cc@example.com')
            //->bcc('bcc@example.com')
            //->replyTo('fabien@example.com')
            //->priority(Email::PRIORITY_HIGH)
            ->subject('Time for Symfony Mailer!')

            // le chemin de la vue Twig à utiliser dans le mail
            ->htmlTemplate('emails/signup.html.twig')

            // un tableau de variable à passer à la vue;
           //  on choisit le nom d'une variable pour la vue et on lui attribue une valeur (comme
dans la fonction `render`) :
            ->context([
                    'expiration_date' => new ¥DateTime('+7 days'),
                    'username' => 'foo',
                ]);

        $mailer->send($email);

        // ...
    }
}
```

Explications :

- ❿ Dans cet exemple la classe `Email` est remplacée par la classe `TemplatedEmail` qui étend de la première mais vient avec des fonctionnalités en plus
- ❿ Une de ces fonctionnalité est la possibilité d'utiliser une vue Twig comme argument de la méthode `htmlTemplate` ('emails/signup.html.twig')
- ❿ Une autre méthode fournie par cette classe est `context()`, qui permet l'envoi de données depuis la fonction définie dans le contrôleur (ou service) vers la page `html.twig` qui sera utilisée
- ❿ L'adresse du destinataire n'est plus écrite sous format chaîne de caractères, mais en utilisant l'objet `Address`

La vue Twig ressemble à ceci :

```
{# templates/emails/signup.html.twig #}
<h1>Welcome {{ email.toName }}!</h1>

<p>
    You signed up as {{ username }} the following email:
</p>
<p><code>{{ email.to[0].address }}</code></p>

<p>
    <a href="#">Click here to activate your account</a>
    (this link is valid until {{ expiration_date|date('F jS') }})
</p>
```

La vue Twig peut accéder non seulement aux paramètres passés à ma méthode `context()` de la classe `TemplatedEmail`, mais aussi à une variable spéciale appelée `email`, qui permet de personnaliser le message (écriture des adresses mail, chemins des images à intégrer etc.).

Ajouter des fichiers joints

Pour joindre des fichiers depuis votre poste à un email, il faut utiliser la méthode `addPart()` et instancier les classes `DataPart` et `File` :

```
 use Symfony¥Component¥Mime¥Part¥DataPart;
 use Symfony¥Component¥Mime¥Part¥File;
 // ...

 $email = (new Email())
     // ...
     ->addPart(new DataPart(new File('/path/to/documents/terms-of-use.pdf')))
     // vous pouvez, si vous le souhaitez, demander aux clients mail d'afficher un certain nom
pour le fichier
     ->addPart(new DataPart(new File('/path/to/documents/privacy.pdf'), 'Privacy Policy'))
     // vous pouvez aussi spécifier le type de document (autrement, il est deviné)
     ->addPart(new DataPart(new File('/path/to/documents/contract.doc'), 'Contract',
'application/msword'))
 ;
```

Note Antérieur à la version 6.2 de Symfony, on pouvait utiliser les fonctions `attachFromPath()` et `attach()` pour joindre des fichiers. Ces méthodes ont été dépréciées et remplacées par `addPart()`.

Intégrer des images

Si vous souhaitez afficher des images dans votre mail, il vaut mieux les intégrer que de les joindre au message. Si vous utilisez la classe `TemplatedEmail`, les images sont incorporées de manière automatique. Autrement, vous devez les intégrer à la main, en faisant appel aux méthodes `addPart` et `asInline()`.

Exemple d'intégration dans un email simple :

```
$email = (new Email())
    // ...
    ->addPart((new DataPart(fopen('/path/to/images/logo.png', 'r'), 'logo',
'image/png'))->asInline())
```

```
    ->addPart((new DataPart(new File('/path/to/images/signature.gif'), 'footer-signature',
'image/gif'))->asInline())

    // utiliser la syntaxe 'cid:' + "nom de l'image intégrée " pour référencer l'image
    ->html('<img src="cid:logo"> ... <img src="cid:footer-signature"> ...')

    // utiliser la même syntaxe pour les images intégrées en tant que background
    ->html('... <div background="cid:footer-signature"> ... </div> ...')
;
```

Dans les emails construits avec la classe `TempaltedEmail`, on peut se servir de la variable spéciale `email` pour intégrer les images. Tout d'abord, il faut définir dans le fichir `config/packages/twig.yaml` un `namespace` appelé par exemple "images" et qui pointe vers votre dossir d'images :

```
# config/packages/twig.yaml
twig:
    # ...

    paths:
        # pointe vers votre dossier d'images
        '%kernel.project_dir%/assets/images': images
```

Ensuite, utilisez la méthode `email.image()` pour intégrer votre image :

```
{# '@images/' c'est la référence au namespece défini dans le fichier de configuration de Twig #}
<img src="{{ email.image('@images/logo.png') }}" alt="Logo">

<h1>Welcome {{ email.toName }}!</h1>
{# ... #}
```

Utiliser le CSS inline

Pour des raisons de sécurité ou par souci de stockage, de nombreux clients mails suppriment les balises du contenu des e-mails, obligeant les développeurs à écrire le style de ces derniers directement dans les balises HTML (`inline`). Pour faciliter ce travail pas simple du tout, Twig fournit une extension appelée `CssInlinerExtension` qui automatise tout pour vous. Pour l'installer, il faut exécuter la commande suivante :

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

Pour l'utliser, il vous faut mettre tout votre code du template de mail à l'intérieur d'un bloc `inline_css` :

```
{% apply inline_css %}
    <style>
        {# définissez votre style CSS comme d'habitude #}
        h1 {
            color: #333;
        }
    </style>

    <h1>Welcome {{ email.toName }}!</h1>
    {# ... #}
```

```
{% endapply %}
```

Note Vous pouvez aussi inclure un fichier css externe! Mais avant de le faire, il vous faut définir un namespece pour votre style dans le fichier de configuration de Twig - comme vous avez procédé pour les images :

```
# config/packages/twig.yaml
twig:
    # ...

    paths:
        # point this wherever your css files live
        '%kernel.project_dir%/assets/styles': styles
```

Maintenant, vous pouvez utiliser votre fichier css (ou vos fichiers, vous pouvez en utiliser plusieurs) :

```
{% apply inline_css(source('@styles/email.css')) %}
    <h1>Welcome {{ username }}!</h1>
    {# ... #}
{% endapply %}
```

Envoyer des messages de manière asynchrone

Quand vous appelez la méthode `send($email)`, l'email est envoyé immédiatement à l'objet de transport (`Symfony` utilise un système basé sur des objets de transport pour envoyer les mails). Mais vous avez aussi la possibilité de stocker les emails dans une file d'attente et de les envoyer plus tard. Et le composant `Messenger` vous permet de le faire.

> Depuis la version 5.4 de Symfony, le composant `Messenger` est activé par défaut et stocke les emails dans la base de données directement.

Afin de pouvoir "consumer" les messages stockés dans la table `messenger_messages`, vous devez démarrer un `worker` (un processus PHP qui exécute du code en mode asynchrone). Pour ce faire, exécutez la commande

```
php bin/console messenger:consume -vv
```

Vous devez choisir les messages que vous voulez envoyer (`async` ou `failed` - les messages qui ont précédemment échoué) et enfin, vous verrez les informations concernant vos mails. S'il n'y a pas d'erreur, la table qui stocke vos messages devrait se vider au fur et à mesure.

```
^Cmihaela@earth:~/Bureau/demo$ php bin/console messenger:consume -vv

 Which transports/receivers do you want to consume?


Choose which receivers you want to consume messages from in order of priority.
Hint: to consume from multiple, use a list of their names, e.g. async, failed

 Select receivers to consume: [async]:
  [0] async
  [1] failed
 > 0


 [OK] Consuming messages from transport "async".

 // The worker will automatically exit once it has received a stop signal via the messenge

 // Quit the worker with CONTROL-C.

21:47:51 INFO      [messenger] Received message Symfony\Component\Mailer\Messenger\SendEma
21:47:51 INFO      [messenger] Message Symfony\Component\Mailer\Messenger\SendEmailMessage
```

Si vous ne voulez pas utilisez pas le composant `Messenger`, désactivez-le en allant dans le fichier `config/packages/messenger.yaml` et en commentant la ligne suivante `Symfony¥Component¥Mailer¥Messenger¥SendEmailMessage: async`.

Pour approfondir le sujet :

- ⓾ le Mailer : https://symfony.com/doc/current/mailer.html#installation
- ⓾ Messenger : https://symfony.com/doc/current/messenger.html#installation

Exercice

Reprenez le formulaire de contact sur lequel vous avez travaillé su le `Form` et, dans la fonction qui traite et valide les données de ce formulaire, ajoutez (si le formulaire est valide) un email en utilisant la classe `TemplatedEmail`.

Les champs du formulaire (adresse mail utilisateur, sujet, message) seront envoyés en tant que variables dans la méthode `context`. Pour le template de mail, vous créerez un dossier `emails` dans le répertoire `templates` et à l'intérieur une vue `contact_email.html.twig`.

Les services en Symfony

Objectifs pédagogiques

A la fin de cette séance vous serez en mesure de :

- ❿ expliquer la notion de `service` dans Symfony
- ❿ utiliser les services de Symfony et en créer d'autres

Introduction

Un `service` est une classe php qui apporte une fonctionalité (une seule tâche bien découpée!) et qui peut être appelée et utilisée un peu partout dans votre application. Une application Symfony contient, dès l'installation, un certain nombre de `services` dont le `Mailer` que nous avons vu précédemment.
Il y a des services pour écrire des messages au journal système (le `Logger`), résoudre une route, valider des données, encoder des mots de passe, accéder à la base de donnée etc.

Et à chaque fois que vous installez un bundle, vous avez accès à d'autres objets de ce type...

Le `Service Container`

Pour accéder à ces outils dans Symfony, il faut les "demander" à un objet spécial appelé `service container` ou `conteneur de service`. C'est ce conteneur de service qui nous permet d'injecter ses services dans nos contrôleurs par exemple, et qui s'assure de leur bonne instanciation (avec les bons paramètres etc.).

Si vous voulez afficher tous les services gérés dans votre application par le `service container`, vous pouvez exécuter la commande suivante :

```
bin/console debug:container
```

```
mihaela@work:~/Bureau/monProjet$ bin/console debug:container
PHP Warning:  Cannot load module "http" because required module "raphf" is not loaded in

Symfony Container Services
==========================

 ------------------------------------------------------------------------------------ ---
  Service ID                                                                          Cl
 ------------------------------------------------------------------------------------ ---
  App\Controller\AccueilController                                                    Ap
  App\Controller\ContactController                                                    Ap
  App\Controller\ProfilController                                                     Ap
  App\DataFixtures\AppFixtures                                                        Ap
  App\DataFixtures\ArtistFixtures                                                     Ap
  App\DataFixtures\DiscFixtures                                                       Ap
  App\DataFixtures\Jeu1                                                               Ap
  App\Entity                                                                          Ap
  App\Form\ContactFormType                                                            Ap
  App\Form\DemoFormType                                                               Ap
  App\Kernel                                                                          al
  App\Repository\ArtistRepository                                                     Ap
  App\Repository\ContactRepository                                                    Ap
  App\Repository\DiscRepository                                                       Ap
  App\Service\MailService                                                             Ap
  Doctrine\Bundle\DoctrineBundle\Controller\ProfilerController                        Do
  Doctrine\Bundle\DoctrineBundle\Dbal\ManagerRegistryAwareConnectionProvider          Do
  Doctrine\Common\Annotations\Reader                                                  al
  Doctrine\Common\Persistence\ManagerRegistry                                         al
  Doctrine\DBAL\Connection                                                            al
  Doctrine\DBAL\Connection $defaultConnection                                         al
  Doctrine\DBAL\Tools\Console\Command\RunSqlCommand                                   Do
  Doctrine\ORM\EntityManagerInterface                                                 al
  Doctrine\ORM\EntityManagerInterface $defaultEntityManager                           al
  Doctrine\Persistence\ManagerRegistry                                                al
  Psr\Cache\CacheItemPoolInterface                                                    al
```

ou encore

`bin/console debug:autowiring`

```
mihaela@work:~/Bureau/monProjet$ bin/console debug:autowiring
PHP Warning:  Cannot load module "http" because required module "raphf" is not loaded in

Autowirable Types
=================

 The following classes & interfaces can be used as type-hints when autowiring:

 App\Kernel (kernel)

 Interface for annotation readers.
 Doctrine\Common\Annotations\Reader (annotations.cached_reader)

 Doctrine\Common\Persistence\ManagerRegistry (doctrine)

 A database abstraction-level connection that implements features like events, transactio
 Doctrine\DBAL\Connection (doctrine.dbal.default_connection)
 Doctrine\DBAL\Connection $defaultConnection (doctrine.dbal.default_connection)

 EntityManager interface
 Doctrine\ORM\EntityManagerInterface (doctrine.orm.default_entity_manager)
 Doctrine\ORM\EntityManagerInterface $defaultEntityManager (doctrine.orm.default_entity_m

 Contract covering object managers for a Doctrine persistence layer ManagerRegistry class
 Doctrine\Persistence\ManagerRegistry (doctrine)

 CacheItemPoolInterface generates CacheItemInterface objects.
 Psr\Cache\CacheItemPoolInterface (cache.app)

 Psr\Container\ContainerInterface $parameterBag (parameter_bag)

 Defines a dispatcher for events.
 Psr\EventDispatcher\EventDispatcherInterface (debug.event_dispatcher)

 Describes a logger instance.
 Psr\Log\LoggerInterface (monolog.logger)
 Psr\Log\LoggerInterface $cacheLogger (monolog.logger.cache)
```

Créer un service et l'utiliser dans un contrôleur Symfony

Vous l'aurez compris, Symfony vient déjà avec un certain nombre de services dans son conteneur.
Mais rien ne vous empêche de créer vos propres services!

Par exemple, supposons que l'on veuille créer notre propre service qui va gérer les envois de mails.

On commence par créer, dans le dossier src, un répertoire appelé Service.

Dans ce répertoire on va créer une classe MailService.php et dans cette classe on va définir une fonction sendMail() qui enverra un email à une adresse pré-définie chaque fois qu'un utilisateur utilise le formulaire de contact. Notre fonction prendra en paramètres l'adresse mail de l'utilisateur, le sujet et le corps du message. Le destinataire sera votre site web (à vous d'écrire l'adresse mail de votre choix).

On va reprendre le contrôleur ContactController et, dans la fonction qui gère le formulaire de contact, au lieu de créer un email après la soumission et la validation du formulaire, on va injecter notre service de mails et on va appeler la fonction sendMail().

De cette manière, non seulemnt on réduit le nombre de lignes de code dans notre contrôleur, mais on isole aussi cette méthode qui pourrait être appelée ailleurs dans le projet.

```
<?php

namespace App¥Controller;

use App¥Entity¥Contact;
use App¥Form¥DemoFormType;
use App¥Form¥ContactFormType;
use App¥Service¥MailService;
use Doctrine¥ORM¥EntityManagerInterface;
use Symfony¥Bridge¥Twig¥Mime¥TemplatedEmail;
use Symfony¥Component¥HttpFoundation¥Request;
use Symfony¥Component¥HttpFoundation¥Response;
use Symfony¥Component¥Mailer¥MailerInterface;
use Symfony¥Component¥Mime¥Email;
use Symfony¥Component¥Routing¥Annotation¥Route;
use Symfony¥Bundle¥FrameworkBundle¥Controller¥AbstractController;

class ContactController extends AbstractController
{
    #[Route('/contact', name: 'app_contact')]
    public function index(Request $request, EntityManagerInterface $entityManager,
MailerInterface $mailer, MailService $ms): Response
    {
        $form = $this->createForm(ContactFormType::class);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {

            //on crée une instance de Contact
            $message = new Contact();
            // Traitement des données du formulaire
            //...
            //persistance des données

            $entityManager->persist($message);
            $entityManager->flush();

            //envoi de mail avec notre service MailService
```

```
            $email = $ms->sendMail('hello@example.com', $message->getEmail(),
$message->getObjet(), $message->getMessage() );
//            dd($message->getEmail());

        }
    }
}
```

Injection de dépendances

Pour utiliser un service dans un autre service, on utilise un procédé appelé injection des dépendances (dependency injection en anglais). C'est-à-dire qu'on injecte dans une méthode __construct() les service dont on a besoin et... c'est tout! Le conteneur se charge de nous mettre à disposition ces classes!

```
use Symfony¥Component¥Mailer¥MailerInterface;

class MailService
{
    private $mailer;

    //On injecte dans le constructeur le MailerInterface

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

//...

}
```

Note

La configuration par défaut des services d'une application Symfony est définie dans le fichier config/services.yaml :

```
# This file is the entry point to configure your own services.
# Files in the packages/ subdirectory configure your dependencies.

# Put parameters here that don't need to change on each machine where the app is deployed
# https://symfony.com/doc/current/best_practices.html#use-parameters-for-application-
configuration
parameters:

services:
    # default configuration for services in *this* file
    _defaults:
        autowire: true      # Automatically injects dependencies in your services.
        autoconfigure: true # Automatically registers your services as commands, event
subscribers, etc.

    # makes classes in src/ available to be used as services
    # this creates a service per class whose id is the fully-qualified class name
    App¥:
        resource: '../src/'
```

```
        exclude:
            - '../src/DependencyInjection/'
            - '../src/Entity/'
            - '../src/Kernel.php'

    # add more service definitions when explicit configuration is needed
    # please note that last definitions always *replace* previous ones
```

Si vous créez un service qui nécessite une configuration particulière, il faudrait le déclarer dans ce fichier !

A part les classes définies comme services, le fichier `services.yaml` contient aussi des "paramètres" de configuration (la ligne `parameters`). Il s'agit de variables dont les valeurs sont utilisées dans plusieurs endroits du projet. Une de ces variables à retenir est `kernel.project_dir`, qui pointe sur le répertoire du projet Symfony. Ce qui veut dire que si nous avons besoin d'accéder à ce répertoire à l'intérieur de nos contrôleurs ou services, nous pouvons appeler ce paramètre.

Par exemple, si on a besoin du même répertoire `/public/assets/images` dans plusieurs contrôleurs ou services (par exemple dans un service qui gère le téléchargement des fichiers), on peut définir un paramètre dans `services.yaml`sous la clé `parameters` :

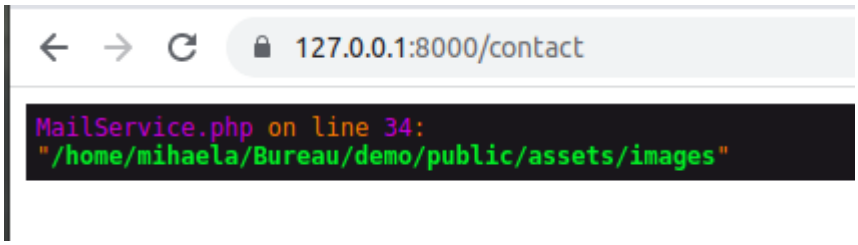
```
# config/services.yaml
parameters:
//on peut lui donnéer le nom qu'on veut, mais c'est ce nom qui va nous aider à récupérer la
valeur de ce paramètre

    images_directory: '%kernel.project_dir%/public/assets/images'
```

Ensuite, il ne nous reste plus qu'à injecter le `parameterBag` pour accéder à ce répertoire dans un service Symfony. Exemple :

```
<?php

namespace App¥Service;
use Symfony¥Component¥DependencyInjection¥ParameterBag¥ParameterBagInterface;
//...

class MailService
{
    //On injecte l'interface ParameterBag

    private $paramBag;
    public function __construct(ParameterBagInterface $paramBag){

        $this->paramBag = $paramBag;
    }

    public function sendMail($expediteur, $destinataire, $sujet, $message){

    //On se sert du parameterBag et du nom du paramètre ('image_directory') pour récupèrer le
chemin du dossier "images"
        $dossiers_images = $this->paramBag->get('images_directory');
    //...
    }
```

}

Si vous faites un `dump and die (dd)` de la variable `$dossier_images`, vous devez avoir le chemin de votre dossier :



Travail à réaliser

Créez la classe `MailService.php` dans `src/Service` et déplacéz-y la fonction d'envoi de mail que vous avez utilisée dans `ContactController`. Ensuite, appelez cette fonction depuis le contrôleur, après la validation du formulaire de contact. Assurez-vous que les mails partent toujours!

> Pour approfondir la notion de `Service` et le `Service Container`, consultez la documentation Symfony sur ce sujet! :
> https://symfony.com/doc/current/service_container.html#fetching-and-using-services

énements Doctrine

Objectifs pédagogiques

À la fin de ce cours, vous devriez être en mesure de :

- ❿ comprendre le concept d'Événements dans Symfony, les notions de `listener` et de `subscriber` (Symfony & Doctrine)
- ❿ créer et utiliser les `EventSubscribers` de Doctrine pour détecter les changements dans la base de données

Prérequis

Avoir une connaissance de base de Symfony et de Doctrine.

Introduction

Comprendre les EventSubscribers

Qu'est-ce qu'un`EventSubscriber` ?

Un `EventSubscriber` (que l'on pourrait traduire par "abonné aux événements") est une classe qui écoute les événements (`Events`) et exécute des actions en réponse à ces événements. Il s'agit d'une fonctionnalité puissante de Symfony pour gérer les événements et effectuer des actions spécifiques.

Événements de Symfony vs Événements de Doctrine :

Bien qu'ils soient souvent utilisés ensemble dans une application Symfony, les événements de Symfony et de Doctrine sont deux systèmes d'événements distincts.

- Les événements de Symfony sont utilisés pour déclencher des actions spécifiques en réponse à des événements survenant dans l'application (événements liés à la gestion des requêtes HTTP (comme `kernel.request), aux formulaires, à la sécurité, aux notifications, etc.).
- Les événements de Doctrine sont spécifiques à cet ORM et sont utilisés pour écouter et réagir aux opérations de la base de données effectuées par Doctrine, telles que la création, la mise à jour, la suppression ou le chargement d'entités. Doctrine émet des événements lors de différentes étapes du cycle de vie d'une entité, par exemple, `postPersist` après l'insertion d'une entité dans la base de données, `preUpdate` avant la mise à jour d'une entité, etc.

Les événements de Doctrine permettent aux développeurs d'intercepter et de réagir à ces opérations de base de données en exécutant des actions supplémentaires. Par exemple, vous pouvez envoyer un e-mail, mettre à jour une autre entité ou effectuer toute autre logique métier nécessaire.

Même si les deux type d'événéments sont distincts, Symfony fournit des moyens d'intégrer les événements de Doctrine dans son système d'événements. Ainsi, vous pouvez créer des EventSubscribers pour écouter à la fois les événements de Symfony et les événements de Doctrine, et agir en conséquence.

EventSubscriber ou EventListener ?

Dans Symfony, vous pouvez utiliser à la fois des listeners ("écouteurs d'événements") et des subscribers en fonction des besoins de votre application. Un EventSubscriber peut écouter **plusieurs événements**, tandis qu'un EventListener écoute **un événement spécifique**. Un subscriber est plus flexible car il peut écouter et réagir à plusieurs événements avec une seule classe.

Les événements de Doctrine les plus couramment utilisés :

- `prePersist` : Cet événement est déclenché juste avant qu'une entité soit persistée pour la première fois dans la base de données. Il peut être utilisé pour effectuer des actions avant l'insertion d'une nouvelle entité, comme la génération de valeurs par défaut, la validation supplémentaire, etc.
- `postPersist` : Cet événement est déclenché après l'insertion d'une entité dans la base de données. Il est utile pour effectuer des actions supplémentaires une fois que l'entité a été persistée, telles que l'envoi d'e-mails, la mise à jour de caches, etc.
- `preUpdate` : Cet événement est déclenché avant la mise à jour d'une entité dans la base de données. Il permet de prendre des mesures avant que les modifications ne soient persistées, comme la validation supplémentaire, la modification d'autres entités liées, etc.
- `postUpdate` : Cet événement est déclenché après la mise à jour d'une entité dans la base de données. Il peut être utilisé pour effectuer des actions supplémentaires après la mise à jour de l'entité, comme l'enregistrement d'audits, la notification des utilisateurs, etc.
- `preRemove` : Cet événement est déclenché avant la suppression d'une entité de la base de données. Il peut être utilisé pour effectuer des actions avant la suppression, telles que la validation, la vérification des dépendances, etc.
- `postRemove` : Cet événement est déclenché après la suppression d'une entité de la base de données. Il est utile pour effectuer des actions supplémentaires après la suppression de l'entité, comme la suppression des fichiers associés, la mise à jour d'autres entités, etc.

⑩ onFlush : Cet événement est déclenché avant la validation et la persistance des objets dans la base de données. Il permet de travailler avec l'ensemble des modifications en cours avant qu'elles ne soient exécutées.

Attention Les événements de Doctrine sont spécifiques à cet ORM et ne réagissent pas aux scripts SQL exécutés directement sur la base de données! Lorsqu'ils sont exécutés, ces scripts SQL ne passent pas par le processus de persistance de Doctrine, et par conséquent, les événements de Doctrine ne sont pas impliqués.

Création d'un EventSubscriber Doctrine

Commençons par la création d'un répertoire EventSubscriber dans le dossier src du projet monProjet. Danc ce nouveau répertoire, on va créer une classe ContactSubscriber.php. Cette classe écoutera les événements de Doctrine pour détecter les nouveaux enregistrements(insert). A la soumission du formulaire de contact que nous avons créé précédemment, on va vérifier que l'objet ou le corps du message contient le mot "RGPD". Si c'est le cas, un email d'alerte sera envoyé à l'administrateur du site velvet.

Dans le fichier src/EventSubscriber/ContactSubscriber.php, ajoutons le contenu suivant :

```
<?php

namespace App¥EventSubscriber;

use App¥Entity¥Contact;
use Doctrine¥Common¥EventSubscriber;
use Doctrine¥ORM¥Events;
use Doctrine¥Persistence¥Event¥LifecycleEventArgs;
use Symfony¥Component¥Mailer¥MailerInterface;
use Symfony¥Component¥Mime¥Email;

class ContactSubscriber implements EventSubscriber
{
    private $mailer;

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

    public function getSubscribedEvents()
    {
        //retourne un tableau d'événements (prePersist, postPersist, preUpdate etc...)
        return [
            //événement déclenché après l'insert dans la base de donnée
            Events::postPersist,
        ];
    }

    public function postPersist(LifecycleEventArgs $args)
    {
//        $args->getObject() nous retourne l'entité concernée par l'événement postPersist
        $entity = $args->getObject();
```

```
   //     Vérifier si l'entité est un nouvel objet de type Contact;
   //    Si l'objet persité n'est pas de type Contact, on ne veut pas que le Subscriber se
déclenche!
           if ($entity instanceof ¥App¥Entity¥Contact) {

               $objet = $entity->getObjet();
               $message = $entity->getMessage();

               //Si l'objet ou le text du message contiennent le mot "rgpd", le Subscriber
enverra un email à l'adresse "admin@velvet.com"
               if (preg_match("/rgpd¥b/i", $objet) || preg_match("/rgpd¥b/i", $message) ) {
                   //     Envoyer un e-mail à l'admin
                   $email = (new Email())
                       ->from('votre_adresse_email@example.com')
                       ->to('admin@velvet.com')
                       ->subject('Alerte RGPD')
                       ->text("Un nouveau message en rapport avec la loi sur les RGPD vous a été
envoyé! L'id du message : " .$entity->getId(). " ¥n Objet du message : " .$entity->getObjet(). "
¥n Texte du message : " .$entity->getMessage());

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

           }
       }
   }
```

Explications :

- ❿ `getSubscribedEvents()` - cette méthode retourne un tableau d'événements auxquels le subscriber doit s'abonner. L'événement est associé à la méthode `postPersist()` dans le Subscriber.
- ❿ Si un nouvel objet de type `Contact` est persisté et que l'objet ou le corps de celui-ci contiennent le mot "RGPD", la fonction déclenché l'envoi d'un email à l'administrateur du site `velvet`.
- ❿ La fonction PHP `preg_match()` est utilisée pour chercher le mot "rgpd" (la regex contient un modificateur `i` qui indique une correspondance insensible à la casse et un `¥b`, qui représente une limite de mot ("word boundary") pour s'assurer que le mot "rgpd" est trouvé comme un mot entier.

Configuration du Subscriber

Mais avant d'utiliser le Subscriber, il faut le déclarer dans le fichier `services.yaml`.

```
services:
......
# add more service definitions when explicit configuration is needed
   App¥EventSubscriber¥ContactSubscriber:
       arguments:
           $mailer: '@Symfony¥Component¥Mailer¥MailerInterface'
       tags:
           - { name: doctrine.event_subscriber }
```

Cette déclaration utilise le tag `doctrine.event_subscriber` pour indiquer à Doctrine que la classe `ContactSubscriber` est un subscriber(un abonné) pour les événements spécifiés dans la méthode `getSubscribedEvents()`. Cela permet à Doctrine de détecter ce subscriber et d'activer les écoutes pour les événements de la base de données correspondants. Elle spécifie les arguments qui seront passés au constructeur de la classe lors de son instanciation (ici, le Mailer)

Testez le Subscriber dans votre projet! Vous pouvez modifier, si vous le souhaitez, l'action à effectuer dans la fonction `postPersist` ou ajouter un autre type d'événement (ex. `preUpdate` etc.)

Liens pour approfondir ce chapitre :

- Événements de Symfony : https://symfony.com/doc/current/event_dispatcher.html
- Évenements de Doctrine : https://www.doctrine-project.org/projects/doctrine-orm/en/current/reference/events.html#the-event-system

L'Authentification dans Symfony

Objectifs pédagogiques

A la fin de cette séance, vous serez capables de mettre en place un système d'authentification dans un projet `Symfony`, en appliquant les bonnes pratiques de sécurité.

Cheminement

Découverte du composant `Security`

La sécurité est un des aspects les plus importants d'une application web et ce n'est pas une chose si simple. Pour faciliter le travail des développeurs, `Symfony` fournit plusieurs outils essentiels via le composant `Security`.

Installation

Pour utiliser le composant `Security` de `Symfony`, il faut d'abord l'installer:

```
composer require symfony/security-bundle
```

Maintenant qu'il est installé, on va pouvoir aborder les trois piliers du système de sécurité dans un projet Symfony :

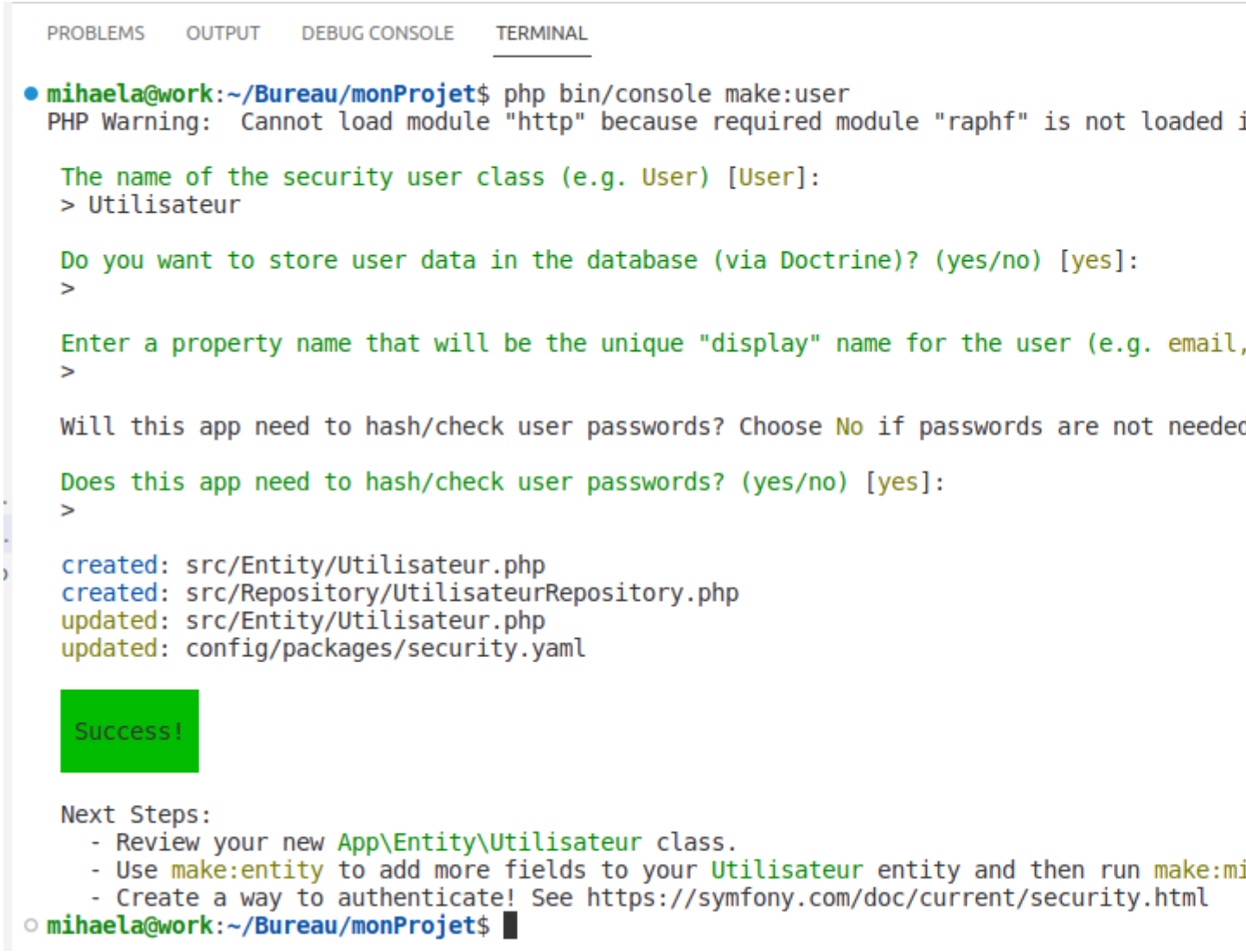
- la classe User (chargée d'inscrire et authentifier les utilisateurs)
- l'authentification (vérification de l'identité du visiteur de votre site - utilisateur anonyme ou authentifié)
- l'autorisation (contrôle d'accès à certaines sections/pages du site)

Création de la classe `User`

Dans un projet Symfony, les permissions sont toujours liées à un objet `user` (utilisateur). Si l'on a besoin de sécuriser certaines sections de notre site, il nous faudrait créer d'abord une classe `User`. On peut la créer avec le maker :

```
php bin/console make:user
```

Le maker suggère `User` comme nom de la classe, mais on peut lui en donner un autre. On va l'appeler `Utilisateur`.



Et voici notre classe :

```
<?php

namespace App¥Entity;

use App¥Repository¥UtilisateurRepository;
use Doctrine¥ORM¥Mapping as ORM;
use Symfony¥Component¥Security¥Core¥User¥PasswordAuthenticatedUserInterface;
use Symfony¥Component¥Security¥Core¥User¥UserInterface;

#[ORM¥Entity(repositoryClass: UtilisateurRepository::class)]
class Utilisateur implements UserInterface, PasswordAuthenticatedUserInterface
{
    #[ORM¥Id]
    #[ORM¥GeneratedValue]
    #[ORM¥Column]
    private ?int $id = null;

    #[ORM¥Column(length: 180, unique: true)]
    private ?string $email = null;
```

```
#[ORM¥Column]
private array $roles = [];

/**
 * @var string The hashed password
 */
#[ORM¥Column]
private ?string $password = null;

public function getId(): ?int
{
    return $this->id;
}

public function getEmail(): ?string
{
    return $this->email;
}

public function setEmail(string $email): self
{
    $this->email = $email;

    return $this;
}

/**
 * A visual identifier that represents this user.
 *
 * @see UserInterface
 */
public function getUserIdentifier(): string
{
    return (string) $this->email;
}

/**
 * @see UserInterface
 */
public function getRoles(): array
{
    $roles = $this->roles;
    // guarantee every user at least has ROLE_USER
    $roles[] = 'ROLE_USER';

    return array_unique($roles);
}

public function setRoles(array $roles): self
{
    $this->roles = $roles;

    return $this;
}
```

```
    /**
     * @see PasswordAuthenticatedUserInterface
     */
    public function getPassword(): string
    {
        return $this->password;
    }

    public function setPassword(string $password): self
    {
        $this->password = $password;

        return $this;
    }

    /**
     * @see UserInterface
     */
    public function eraseCredentials()
    {
        // If you store any temporary, sensitive data on the user, clear it here
        // $this->plainPassword = null;
    }
}
```

Explications :

⑩ Contrairement aux autres entités que vous avez créées, l'entité utilisateur implémente deux classes liées au système d'authentification - `UserInterface` et `PasswordAuthenticatedUserInterface`.

⑩ Ces interfaces permettent, entre autres, la gestion de la propriété unique (`userIdentifier`, par défaut c'est l'adresse mail), des rôles (permissions) et du hashage du mot de passe.

⑩ Quand vous utilisez la commande `make:user`, cette classe apparaît comme `UserProvider` (classe responsable de la gestion des utilisateurs) dans le fichier `security.yaml`.

```
#config/packages

security:
    # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
    password_hashers:
        Symfony¥Component¥Security¥Core¥User¥PasswordAuthenticatedUserInterface: 'auto'
    # https://symfony.com/doc/current/security.html#loading-the-user-the-user-provider
    providers:
        # used to reload user from session & other features (e.g. switch_user)
        app_user_provider:
            entity:
                class: App¥Entity¥Utilisateur <--- votre UserProvider
                property: email <--- l'identifiant unique
```

Remarquez l'entrée `password_hashers` avant `providers` : dès la création de la classe, Symfony vous permet d'utiliser le service `UserPasswordHasherInterface` pour hasher le mot de passe de vos utilisateurs avant de les insérer dans la base de donnée. Par défaut, Symfony vous propose "auto" pour le système de hashage, ce qui veut dire que

vous lui laissez la responsabilité de choisir la fonction de hashage la plus récente (aujourd'hui bcrypt, mais il y en a d'autres).

Voici comment hasher le mot de passe dans un contrôleur :

Vous allez pouvoir créer ce formulaire d'inscription avec la commande make:registration-form. Le maker vous créera le contrôleur RegistrationController.php, le FormType-Form/RegistrationFormTypè et le formulaire d'inscription -templates/registration/register.html.twig`.

```
// src/Controller/RegistrationController.php
namespace App¥Controller;

// ...
use Symfony¥Component¥HttpFoundation¥Response;
use Symfony¥Component¥PasswordHasher¥Hasher¥UserPasswordHasherInterface;

class RegistrationController extends AbstractController
{
    public function index(UserPasswordHasherInterface $passwordHasher): Response
    {
        // ... e.g. get the user data from a registration form
        $user = new User(...);
        $plaintextPassword = ...;

        // hash the password (based on the security.yaml config for the $user class)
        $hashedPassword = $passwordHasher->hashPassword(
            $user,
            $plaintextPassword
        );
        $user->setPassword($hashedPassword);

        // ...
    }
}
```

Vous pouvez aussi hasher le mot de passe en ligne de commande (pour des tests, ne le faites pas en production!) : php bin/console security:hash-password!

Le Firewall

Votre fichier security.yaml contient une troisième entrée (après password_hashers et providers) :

```
firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: app_user_provider
```

La section appelée firewalls constitue le centre névralgique de votre application Symfony : c'est elle qui gère l'authentification! Le firewall(en français pare-feu) définit les espaces protégés par

un login et la manière dont les utilisateurs peuvent se connecter (formulaire de login, jeton API, Oauth etc.).

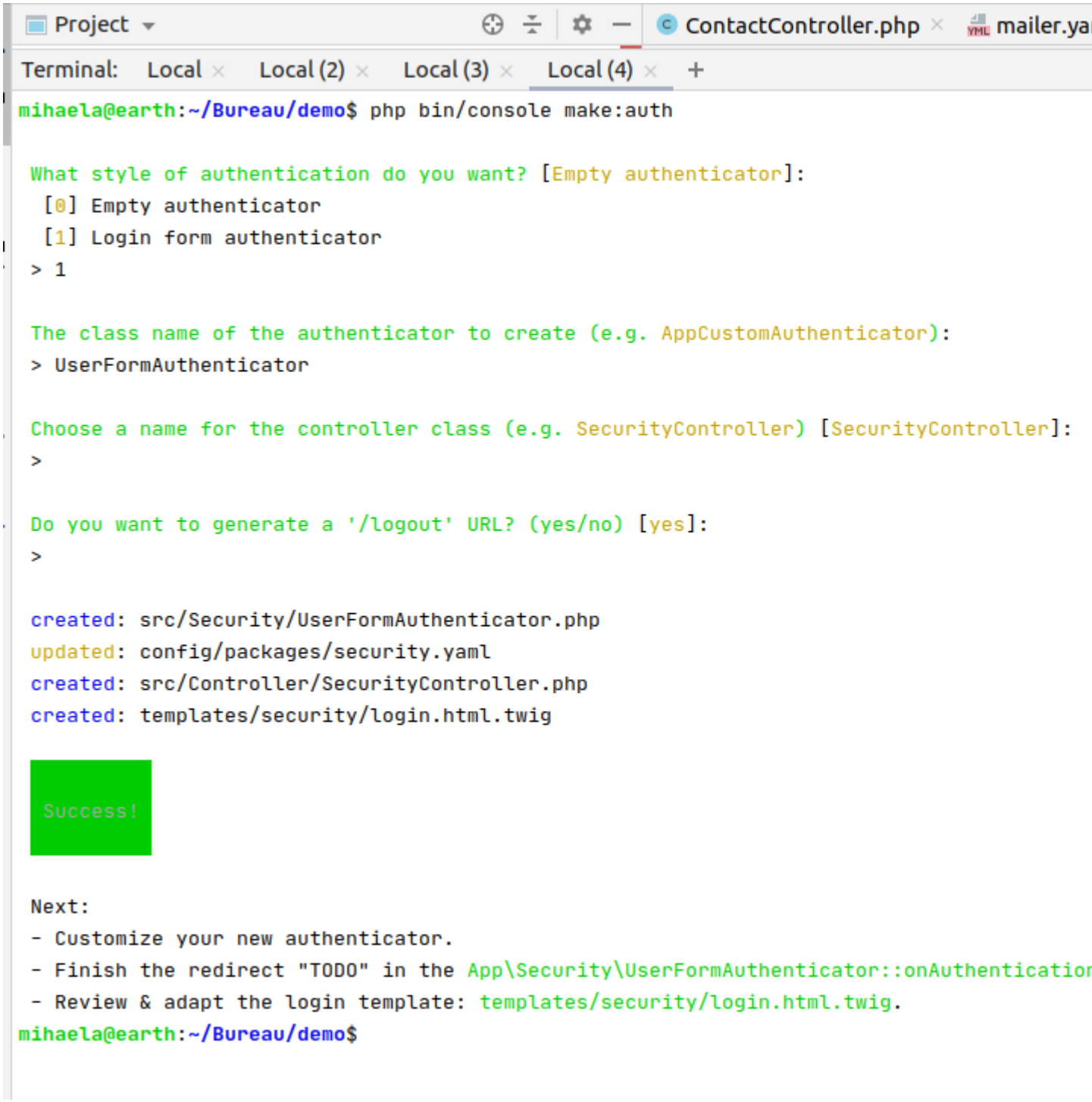
Le "main" est le `firewall` principal (il gère tous les URLs, c'est pratiquement l'entrée de votre site!). Si vous en rajoutez d'autres (un firewall appelé `api`, par exemple), vous devez les déclarer **avant** le `main`!

Authentification

Le formulaire de login La plupart des sites web ont un formulaire de login qui permet aux utilisateurs de s'authentifier à l'aide d'un identifiant (email ou username) et un mot de passe. Dans Symfony, on peut créer ce formulaire avec la commande :

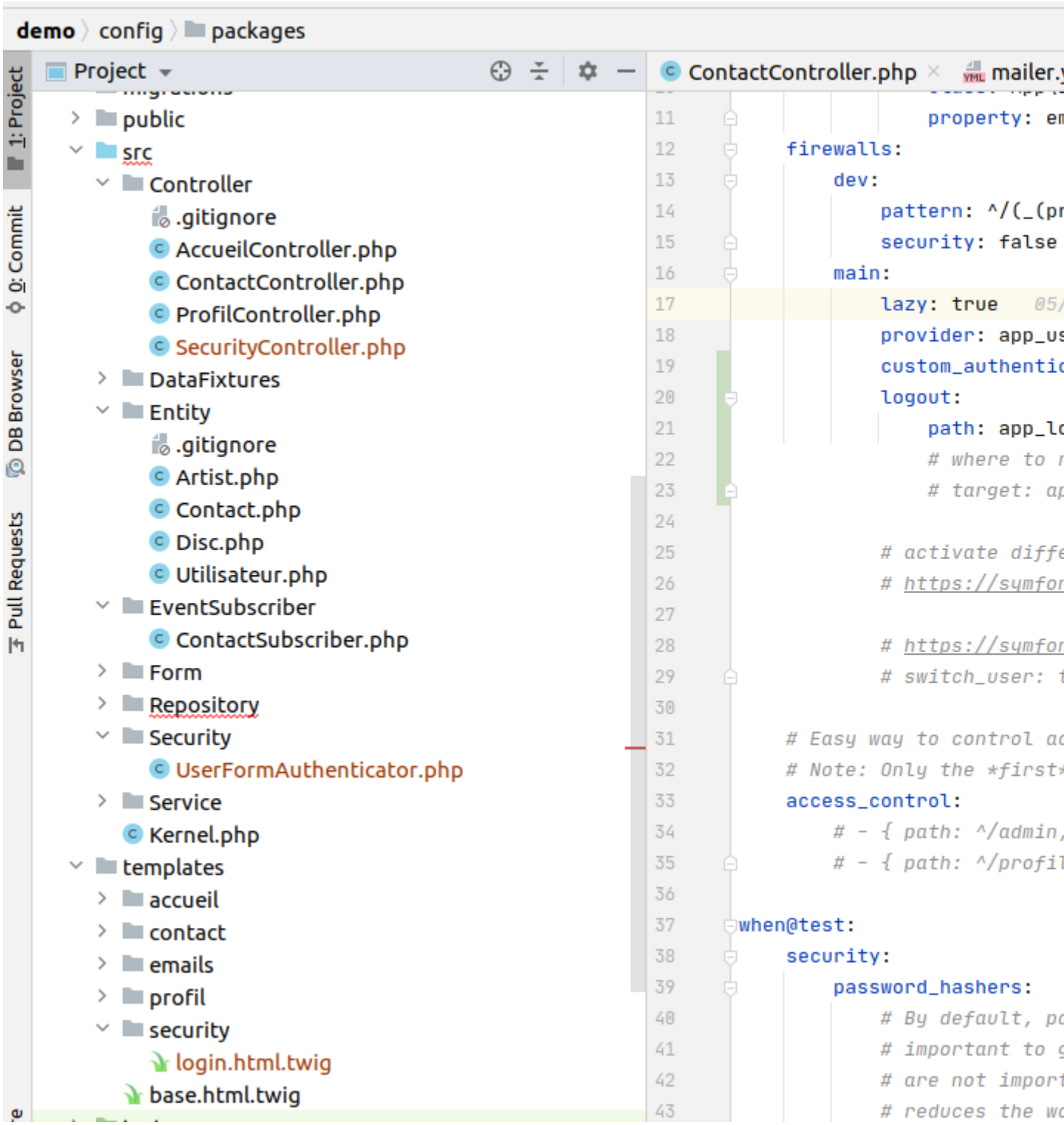
```
php bin/console make:auth
```

⑩ crée un contrôleur, un formulaire et une classe chargée de gérer l'authentification par formulaire de login (l'`Authenticator`)



Et voilà! Votre projet contient de nouveaux fichiers - `SecurityController.php`,

src/Security/UserFormAuthenticator.php et templates/security/login.html.twig).



Ouvrez le fichier UserFormAuthenticator.php. On va modifier la fonction onAuthenticationSuccess afin de lui rajouter la route vers laquelle l'utilisateur sera redirigé après connexion :

```
    public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $firewallName): ?Response
    {
        //s'il y a une route dans la session avant login, l'utilisateur sera redirigé vers cette route
```

```
        if ($targetPath = $this->getTargetPath($request->getSession(), $firewallName)) {
            return new RedirectResponse($targetPath);
        }

        // sinon, on l'envoie sur la page `profil`
        return new RedirectResponse($this->urlGenerator->generate('app_profil'));
        throw new ¥Exception('TODO: provide a valid redirect inside '.__FILE__);
    }
```

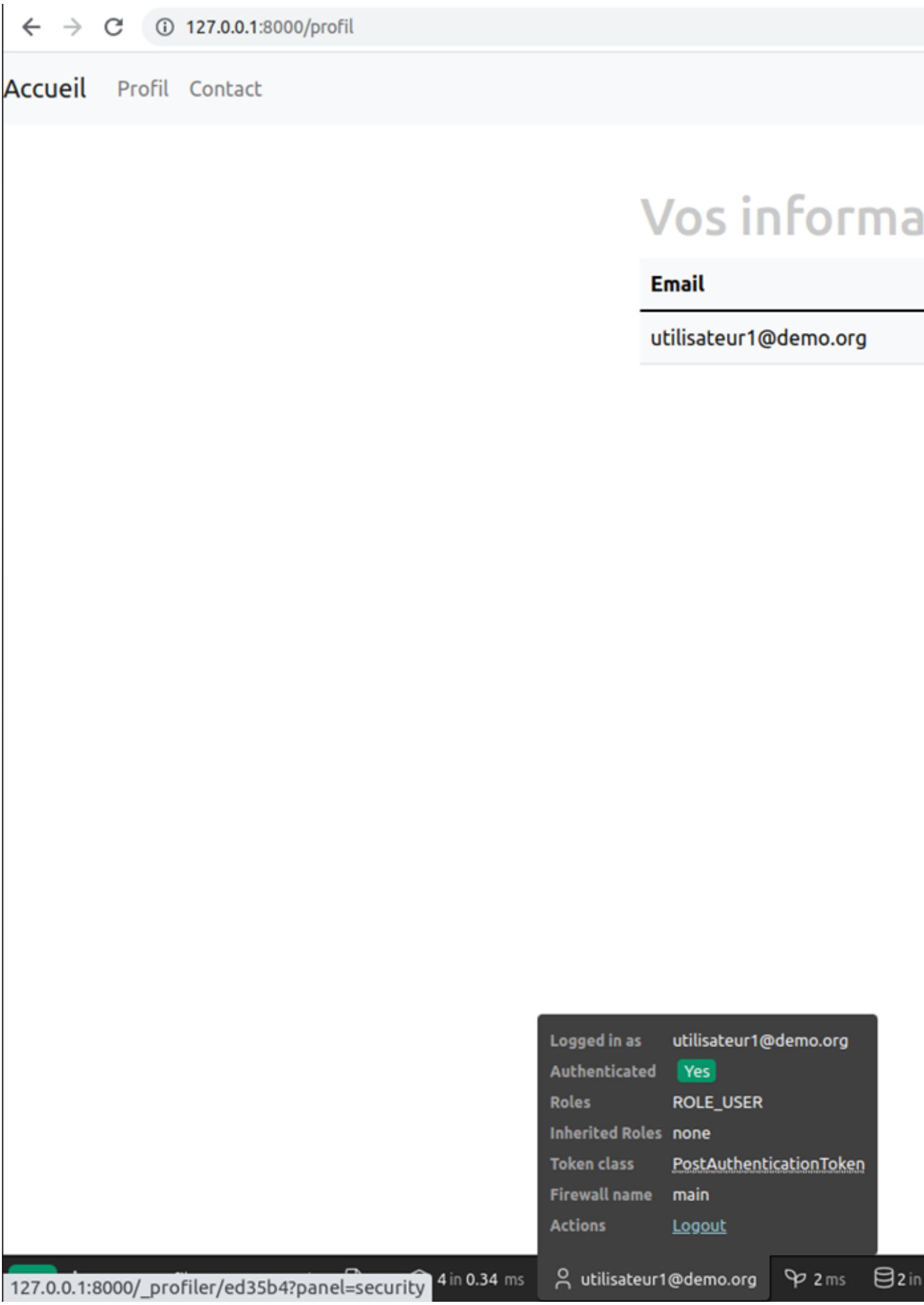
Créons maintenant un utilisateur en utilisant le formulaire d'inscription (chargé par la route `/register`). Dans le contrôleur `ProfilController`, on va modifier le code pour pouvoir afficher les informations de cet utilisateur une fois qu'il sera connecté :

```
<?php

namespace App¥Controller;

use App¥Repository¥UtilisateurRepository;
use Symfony¥Bundle¥FrameworkBundle¥Controller¥AbstractController;
use Symfony¥Component¥HttpFoundation¥Response;
use Symfony¥Component¥Routing¥Annotation¥Route;

class ProfilController extends AbstractController
{
    private $userRepo;

    public function __construct(UtilisateurRepository $userRepo){
        $this->userRepo = $userRepo;
    }
    #[Route('/profil', name: 'app_profil')]
    public function index(): Response
    {
        $identifiant = $this->getUser()->getUserIdentifier(); <-- ICI on récupère l'identifiant
unique de l'utilisateur connecté
        if($identifiant){
            $info = $this->userRepo->findOneBy(["email" =>$identifiant]); <--- ICI on vérifie
qu'on a bien un utilisateur dans la base de donnée qui a ce mail
        }

        return $this->render('profil/index.html.twig', [
            'informations' => $info
        ]);
    }
}
```

Ensuite, dans la page `templates/profil/index.html.twig`, on va afficher le mail de l'utilisateur :

```
{% extends 'base.html.twig' %}

{% block body %}
    <div class="container">
        <div class="row mt-5">
            <div class="col-sm-8 offset-sm-2">
                <h1>Vos informations</h1>
                <table class="table table-light table-hover">
```

```
                    <thead>
                    <tr>
                        <th>Email</th>
                    </tr>
                    </thead>
                    <tbody>
                    <tr>
                        <td>{{ informations.email }}</td>
                    </tr>
                    </tbody>
                </table>

            </div>
        </div>
    </div>
{% endblock %}
```


127.0.0.1:8000/profil
Accueil
Profil
Contact
Vos informa
Email
utilisateur1@demo.org
Logged in as
utilisateur1@demo.org
Authenticated
Yes
Roles
ROLE_USER
Inherited Roles
none
Token class
PostAuthenticationToken
Firewall name
main
Actions
Logout
127.0.0.1:8000/_profiler/ed35b4?panel=security
4 in 0.34 ms
utilisateur1@demo.org
2 ms
2 in

Remarques

- Regardez les informations affichées dans la barre de débogage de `Symfony`, en bas de la page : nous avons bien un utilisateur authentifié !
- On aurait pu afficher les mêmes informations en utilisant la variable globale de Twig `app`! Il s'agit d'un objet de contexte injecté par `Symfony` dans chaque template et qui vous permet d'accéder à certaines informations concernant votre application :
- `app.user` - retourne l'utilisateur connecté ou `null` si l'utilisateur n'est pas connecté
- `app.request` - retourne l'objet `Request`
- `app.session` - retourne l'objet `Session` - la session de l'utilisateur connecte ou null s'il n'y a pas d'utilisateur connecté
- `app.flashes` - retourne un tableau de messages flash stockés dans la session.

Vous trouverez dans la documentetion de Symfony plus d'informations sur la variable globale `app`.

On a bien la classe `User`, les formulaires d'inscription et de connexion, le pare-feu principal... On peut donc passer à l'étape suivante : l'autorisation. Autrement dit, qu'est-ce qu'un utilisateur a le droit de voir/faire sur notre site ?

Consulter ce support pour apprendre comment sont gérés dans `Symfony` les droits d'accès.