Sécurité et bonnes pratiques de l'application Green Fit

Voici les principales mesures de sécurité de l'application.

Le bundle SecurityBundle de Symfony fournit une grande panoplie d'outils pour gérer et renforcer la sécurité de l'application.

Pour l'installer : `composer require symfony/security-bundle`

1. L'authentification

Dans ce projet, l'authentification consiste à lier une adresse e-mail servant le login et un mot-de-passe, à une entité existante en base de données, pour accéder à l'administration de l'application.

Mots de passe chiffrés :

Renforcement de la politique de sécurité lors de la création de mot de passe par l'utilisateur.

Les mots de passe sont chiffrés en base de données avec la fonction de hachage bCrypt.

```
12  security:
13      # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
14      password_hashers:
15          App\Entity\User: 'auto'
16          Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface:
17              algorithm: 'auto'
```

Fichier security.yaml

```php
#[Route('/new', name: 'app_users_new', methods: ['GET', 'POST'])]
public function new(Request $request, UsersRepository $usersRepository, UserPasswordHasherInterface
$rolesUsersRepository): Response
{
    $user = new Users();
    $form = $this->createForm(UsersType::class, $user);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        // on fixe automatiquement le rôle ADMIN pour les nouveaux techniciens créés
        $r[]='ROLE_ADMIN';
        $user->setRoles($r);
        $user->setRolesUsers($rolesUsersRepository->find(1));
        // encodage du mote de passe
        $user->setPassword(
            $userPasswordHasher->hashPassword(
                $user,
                $form->get('password')->getData()
            )
        );
```

Encodage du mot de passe dans le fichier UsersController.php

id	email	roles (DC2Type:json)	password	lastname	firstname
1	admin@greenfit.fr	["ROLE_ADMIN"]	$2y$13$Ee9JW95GDnqzvkmwMBE4h.sEi4OpUtaEXtijTTrz5Zk...	Ollivier	Jérôme
7	orleans@greenfit.fr	["ROLE_PARTNER"]	$2y$13$mZkeEOxclJ6Cxr.r6kEW7eTeBtEnnx1hFHKDfGUGmGj...	Dubois	Paul
8	orleans-bannier@greenfit.fr	["ROLE_STRUCTURE"]	$2y$13$acZYu/f7rsMX5Fajz3T6iesLePn.n1Ho.SipHYyJalM...	Dubet	Chantale
27	loire.orleans@greenfit.fr	["ROLE_STRUCTURE"]	$2y$13$GLCiJvZP2PiyjqaW9t.dpOZb3k6aUYjTkmHJ1TOd.2v...	Martinet	José

Mots de passe encryptés dans la base de données

2. Sécurité et utilisateurs

Contrôle d'accès :

Une fois authentifié, l'utilisateur peut accéder à des fonctionnalités proposées par l'application. Avant cela, il faut contrôler s'il a le droit d'y accéder. En utilisant le contrôle d'accès et le vérificateur d'autorisation, je contrôle les permissions requises pour effectuer une action spécifique ou visiter une URL spécifique.

```yaml
56      access_control:
57
58          - { path: ^/admin, roles: ROLE_ADMIN }
59          - { path: ^/modules, roles: ROLE_ADMIN }
60          - { path: ^/users, roles: ROLE_ADMIN }
61          - { path: ^/partners, roles: [ROLE_ADMIN, ROLE_PARTNER] }
62          - { path: ^/structures, roles: [ROLE_ADMIN, ROLE_PARTNER, ROLE_STRUCTURE] }
63      role_hierarchy:
64          ROLE_ADMIN: [ROLE_PARTNER, ROLE_STRUCTURE]
```

Contrôle d'accès dans le fichier security.yaml

Exemple de test unitaire pour voir si on peut se connecter à la page /users sans être connecté :

```php
<?php

namespace App\Tests\Controller;

use App\Repository\UsersRepository;
use Symfony\Bundle\FrameworkBundle\Test\WebTestCase;
use Symfony\Component\HttpFoundation\Response;

class UsersPageTest extends WebTestCase
{
    public function testReservationPageIsRestricted()
    {
        $client = static::createClient();
        $client->request('GET', '/users');
        $this->assertResponseRedirects();
    }
}
```

Code de tests/UsersPageTest

Résultat du test : impossible de se connecter à cette page, sans avoir les autorisations pour s'y connecter. Ici il faudrait avoir le rôle `ROLE_ADMIN` pour cela.

```
PS C:\Users\spire\Desktop\Versionning ECF\160922-08h45\xampp\apps\greenfit-ecf> symfony php bin/phpunit --testdox
PHPUnit 9.5.23 #StandWithUkraine

Testing
Url Login (App\Tests\Controller\UrlLogin)
 ✓ Login

Users Page (App\Tests\Controller\UsersPage)
 ✓ Reservation page is restricted

Time: 00:00.240, Memory: 28.00 MB

OK (2 tests, 2 assertions)
```

Utilisateur :

Les permissions dans Symfony sont toujours liées à un objet utilisateur. Pour sécuriser ceratines parties de mon application, j'ai dû créer utilisateur (Users). Il s'agit d'une classe qui implémente `UserInterface`. Il s'agit souvent d'une entité (entity) Doctrine, mais dans ce projet j'ai utilisé la classe User (`php bin console:make user`).

Ce fournisseur d'utilisateurs sait comment charger des utilisateurs à partir d'un stockage (par exemple ici une base de données) en se basant sur un « identifiant utilisateur » (par exemple l'adresse email ou le nom d'utilisateur de l'utilisateur). La configuration utilise Doctrine pour charger l'entité `Users` en utilisant la propriété email comme « identifiant utilisateur ».

```yaml
        app_user_provider:
            entity:
                class: App\Entity\Users
                property: email
```

Fichier security.yam

Politique d'accès aux pages avec le pare-feu :

Grâce au pare-feu, chaque demande est vérifiée si elle nécessite un utilisateur authentifié. Le pare-feu se charge également d'authentifier cet utilisateur (par exemple, à l'aide d'un formulaire de connexion) ;

Au niveau du front-end, les pages du back-end ne sont accessibles qu'aux utilisateurs ayant le rôle « ROLE_ADMIN ».

```yaml
24      firewalls:
25          dev:
26              pattern: ^/(_(profiler|wdt)|css|images|js)/
27              security: false
28          main:
29              lazy: true
30              provider: app_user_provider
31              login_throttling:
32                  max_attempts: 3
33                  interval: '5 minutes'
34              form_login:
35                  login_path: login
36                  check_path: login
37                  enable_csrf: true
38                  default_target_path: app_admin
39                  always_use_default_target_path: true
40
41              logout:
42                  path: logout
43                  target: /
```

Réglages du pare-feu de l'application dans le fichier security.ya

3. Gestion sécurisée des sessions : la protection CSRF dans le formulaire de connexion

Les attaques CSRF de connexion peuvent être évitées en utilisant la même technique d'ajout de jetons CSRF cachés dans les formulaires de connexion. Le composant Sécurité fournit déjà une protection CSRF (Cross Site Request Forgeries), mais j'ai dû configurer certaines options avant de l'utiliser.

Le jeton CSRF est une protection qui requiert l'insertion d'une valeur aléatoire, composé de nombres et de lettres et dynamique dans une requête. Cette valeur est ensuite analysée par le serveur pour déterminer si la requête est légitime

J'ai activé tout d'abord CSRF sur le formulaire de connexion :

```yaml
34              form_login:
35                  login_path: login
36                  check_path: login
37                  enable_csrf: true
```

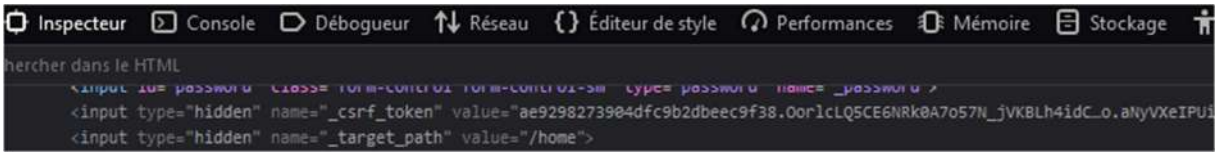
Fichier security.yaml

Puis j'ai utilisé la fonction `csrf_token()` dans le modèle Twig pour générer un jeton CSRF et le stocker dans un champ caché du formulaire. Par défaut, le champ HTML doit être appelé `_csrf_token` et la chaîne utilisée pour générer la valeur doit être `authenticate`.

```
<div class="form-group">
    <label for="password" class="text-light">Mot de passe:</label>
    <input type="password" id="password" name="_password"  class="form-control form-control-sm"/>
    <input type="hidden"  name="_csrf_token" value="{{ csrf_token('authenticate') }}">
```

Fichier templates/login/index.html.twig

À chaque rafraîchissement de la page de login, le CSRF change.



4. Protection contre les injections SQL

Validation obligatoire des données entrées par l'utilisateur dans les formulaires, grâce au système de « contrainte » de Symfony

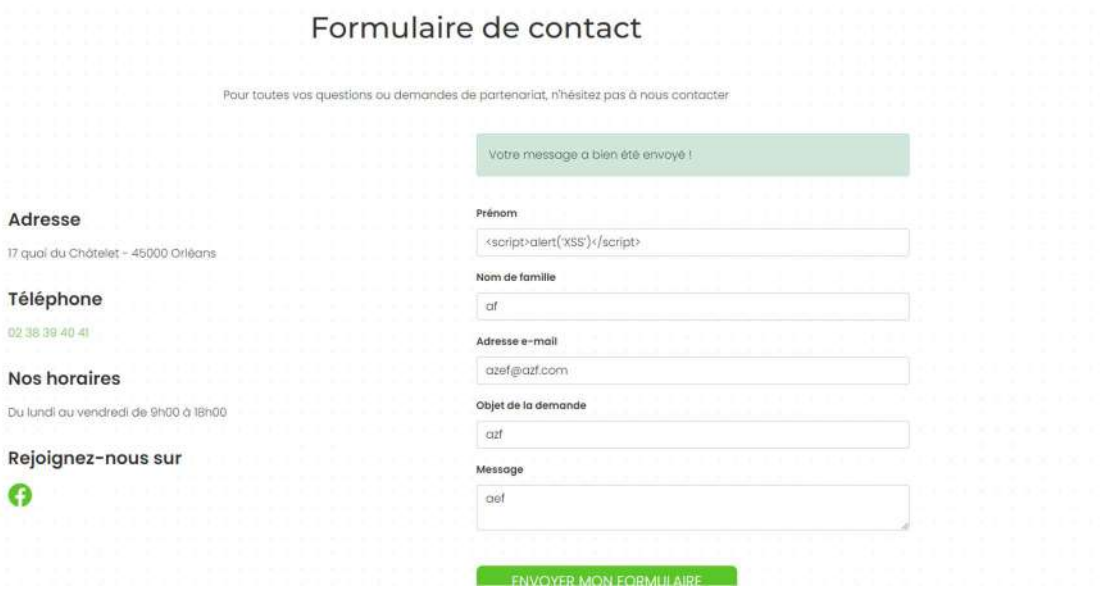
L'ORM Doctrine permet de gérer les interactions avec la base de données et de protéger l'application des injections SQL.

Le moteur de templates TWIG protège l'application contre les attaques de type « Cross-Script Scripting » (XSS) qui sont des failles de sécurité qui permettent à un attaquant d'injecter dans une application web un code client malveillant.

J'ai testé en écrivant un simple script dans un des champs du formulaire de la page de contact :

```
<script>alert('XSS')</script>
```

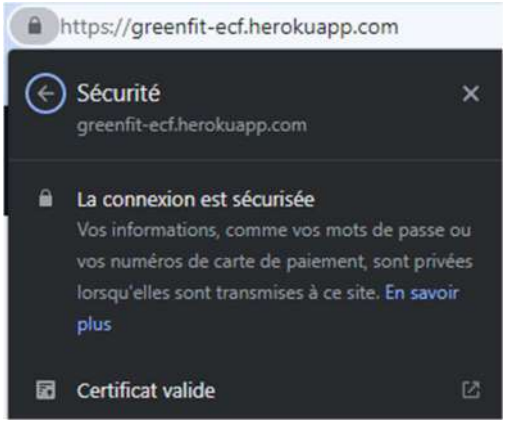
Résultat : le script n'a pas été exécuté. Aucun popup ne s'est affiché avec le message.



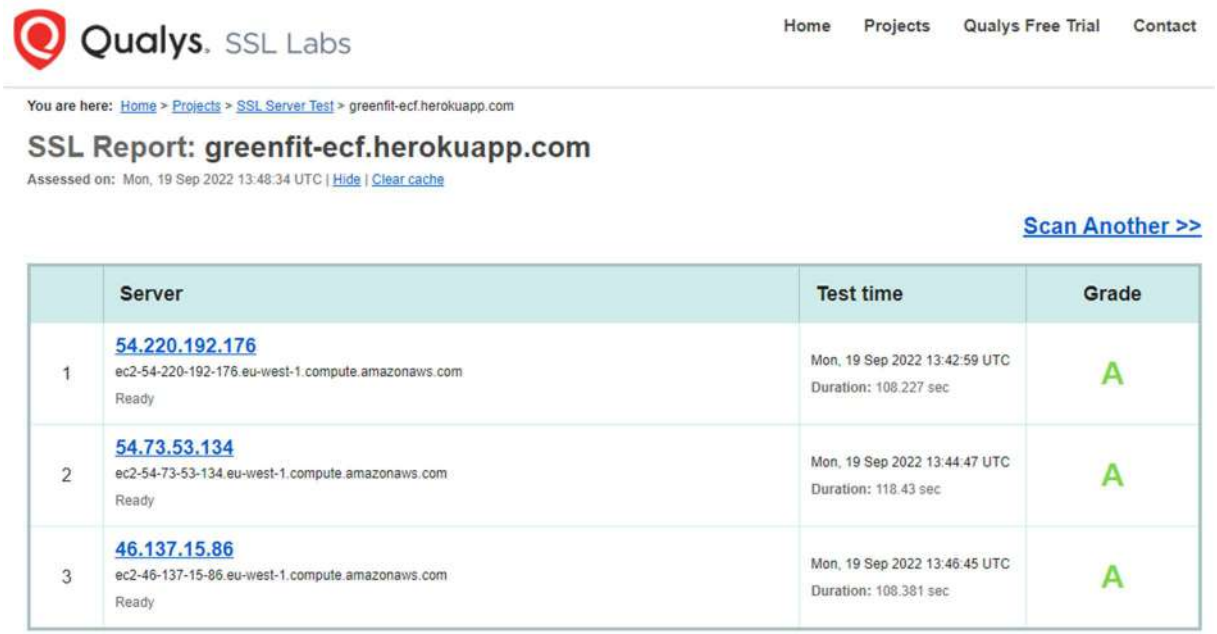
5. Sécurisation des communications avec SSL

Le protocole HTPS (Hypertext Transfer Protocol Secure ou protocole de transfert hypertexte sécurisé) est un protocole de communication Internet qui protège l'intégrité ainsi que la confidentialité des données lors du transfert d'informations entre l'ordinateur de l'internaute et le site

Le service HTTPS activé sur Heroku, vient renforcer la sécurité des échanges, grâce à un certificat SSL (Security Socket Layer, permettant l'authentification, le chiffrement et le déchiffrement des données envoyées.



Certificat SSL



Résultat : l'application bénéficie bien d'un certificat SSL.

6. Limiter le nombre de tentatives de connexion

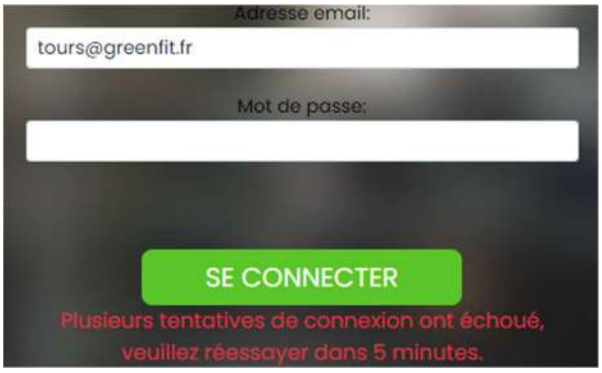
Paramétrage du Login Throttling dans Symfony Grâce à l'ajout du composant RateLimiter depuis Symfony 5.2, je peux bénéficier d'un « Rate Limiter »

https://symfony.com/blog/new-in-symfony-5-2-login-throttling

```
28      main:
29          lazy: true
30          provider: app_user_provider
31          login_throttling:
32              max_attempts: 3
33              interval: '5 minutes'
```

Paramétrage du Login Throttling dans le fichier config/packages/security.yaml

Ici après 3 tentatives d'authentification échouées, l'utilisateur a un message d'erreur, lui bloquant l'authentification dans le formulaire de connexion pendant 5 minutes.



Message d'erreur en cas de trop de tentatives échouées

7. Validation des formulaires avec le Bundle Validator

Par l'installation du bundle Validator, les formulaires sont validés avant l'envoie en base de données, grâce aux contraintes que l'on peut leur appliquer. En cas de non validation, l'utilisateur connecté verra un message s'afficher, lui expliquant pourquoi son formulaire n'a pu être envoyé.

Le validateur est conçu pour valider les objets par rapport à nos contraintes.

Exemple : *pour la création d'un technicien, les contraintes pour le mot de passe sont les suivantes* :

- « `NotBlank` » : le mot de passe ne doit pas être vide
- « `Length` » : le mot de passe doit être compris entre 8 et 4096 caractères

```
'constraints' => [
    new NotBlank([
        'message' => 'Veuillez entrer un mot de passe',
    ]),
    new Length([
        'min' => 6,
        'minMessage' => 'Votre mot de passe doit être au moins de {{ limit }} caractères',
        // max length allowed by Symfony for security reasons
        'max' => 4096,
    ]),

],
```

Fichier src/Form/UsersType.php

Si le technicien rentre un mot de passe trop court, il aura un message le prévenant que son mot de passe doit contenir au minimum 8 caractères, ce qui bloquera la création du technicien tant que le mot de passe ne contiendra pas le nombre minimal de caractère.

Ainsi, dans notre exemple, lorsque le technicien soumettra le formulaire, si l'adresse mail ne comporte pas de 8 à 4096 caractères et/ou si le mot de passe est vide, alors la variable isValid du contrôleur UsersController sera à false, le formulaire sera renvoyé à nouveau, mais cette fois-ci avec les messages d'erreurs de validation.

Message d'erreur à la création d'un nouveau technicien