DOCUMENTATION TECHNIQUE
TODO & CO

Sommaire :

Présentation du projet :

Le site ToDo&Co permet aux utilisateurs inscrit de créer des tâches et de les lister. C'est un site qui est un outil de gestion de projet. Ici les utilisateurs peuvent voir toutes les tâches créées et les modifier ou les marquées comme terminées au besoin mais seul celui qui a crée la tâche peut la supprimer. Le site est relativement jeune et encore en phase de construction, il est donc important de developper un site flexible qui puisse s'adapter aux developpement de nouvelles fonctionnalités. Lors du premier developpement du site des erreurs ont été relevées, un audit a été realisé en même temps que la redaction de cette documentation afin de lister les erreurs et montrer les modifications apportées. Je vous invite à prendre connaissance de cet audit.

Technologies utilisées :

Le site a été mise a jour sur la version 6 du framework Symfony et la version 8 de PHP (plus exactement la 8.0.14). Je conseille de suivre à jour l'évolutions de technologie et de mettre à jour le site regulierement, il est cependant recommandé d'ancrer la mise à jour sur une version longue de Symfony que l'on appelle LTS (long term release). Afin de vous aider dans cette tache, rendez-vous sur cette page :
https://symfony.com/releases
Et choisissez la version «Recommended for most users».

Installation du projet :

Pour installer le projet effectué la commande :

```
Ikan Hiu@Suranadi MINGW64 ~/Desktop/Symfony/P8-ToDoApp (deva)
$ git clone https://github.com/Pierre-Ka/ToDo-OP8.git
```

Puis mettez à jour le .env et la base de donnée comme indiquer dans le ReadMe.

Librairies utilisées :

```php
return [
    Symfony\Bundle\FrameworkBundle\FrameworkBundle::class => ['all' => true],
    Doctrine\Bundle\DoctrineBundle\DoctrineBundle::class => ['all' => true],
    Doctrine\Bundle\MigrationsBundle\DoctrineMigrationsBundle::class => ['all' => true],
    Symfony\Bundle\DebugBundle\DebugBundle::class => ['dev' => true],
    Symfony\Bundle\TwigBundle\TwigBundle::class => ['all' => true],
    Symfony\Bundle\WebProfilerBundle\WebProfilerBundle::class => ['dev' => true, 'test' => true],
    Twig\Extra\TwigExtraBundle\TwigExtraBundle::class => ['all' => true],
    Symfony\Bundle\SecurityBundle\SecurityBundle::class => ['all' => true],
    Symfony\Bundle\MonologBundle\MonologBundle::class => ['all' => true],
    Symfony\Bundle\MakerBundle\MakerBundle::class => ['dev' => true],
    Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle::class => ['all' => true],
    Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle::class => ['dev' => true, 'test' => true],
];
```

L'application possède un certains nombre de librairies installées, lors de l'installation du projet vous devez les installer en effectuant la commande :

```
Ikan Hiu@Suranadi MINGW64 ~/Desktop/Symfony/P8-ToDoApp (deva)
$ composer install
```

Rendez-vous dans le ReadMe pour plus d'informations concernant l'installation. Cette commande installe tous les packages, ces packages évolue avec le temps, il est necessaire d'effectue regulierement des mises à jour avec la commande :

```
Ikan Hiu@Suranadi MINGW64 ~/Desktop/Symfony/P8-ToDoApp (deva)
$ composer update
```

Il se peut egalement qu'avec le temps certains packages installées soient obsolètes et même plus utilisés dans le code. Vous pouvez les identifier avec la commande :

```
Ikan Hiu@Suranadi MINGW64 ~/Desktop/Symfony/P8-ToDoApp (deva)
$ php bin/composer-unused
```

Et les supprimer avec la commande : composer remove.

Securité :

Le projet utilise le composant Sécurité de Symfony, nous allons ici voir les differents aspects de celui-ci et comment le modifier pour repondre à de futurs besoins. Afin d'activer ce composant il faut mettre le

enable_authenticator_manager à true dans config/package/security.yaml security:

```yaml
security:
    enable_authenticator_manager: true

```

Le composant Sécurité de Symfony s'effectue via un formulaire de connexion, l'utilisateur tentant de se connecter et alors comparer à l'utilisateur enregistré en base de donnée. Ce systeme est sécurisé par un mot de passe crypté. Si l'authentification est réussi l'utilisateur est a accès à une partie du site en fonction de son rôle. Nous allons voir ce mécanisme dans le détail.

Cryptage :

Le cryptage s'effectue automatiquement par Symfony lors de la connexion, il est possible de choisir l'algorithme de cryptage dans le même fichier config/package/security.yaml security:

```yaml
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
        App\Entity\User:
            algorithm: auto

```

Ici la strategie adoptée a consisté à garder l'algorithme par défault de Symfony. Dans le cas de la création d'un utilisateur ou le cas de la modification de son mot de passe, se sera à nous de crypter (on dit egalement hasher) le mot de passe, dans ce cas-là il faudra appeler dans la classe en question un objet issu de l'interface UserPasswordHasherInterface et utilisé la methode hashPassword() :

```php
    public function __construct(EntityManagerInterface $em, UserPasswordHasherInterface $hasher)
    {
        $this->em = $em;
        $this->hasher = $hasher;
    }

    public function new($form, $user): void
    {
        $plainPassword = $form->get('password')->getData();
        $password =  $this->hasher->hashPassword($user, $plainPassword);
```

Provider :

Afin de comparer le données soumises dans le formulaire de connexion avec quelque chose, le composant Sécurité de Symfony a besoin que l'on lui définisse un provider. On definit notre provider dans le fichier config/package/security.yaml security:

```yaml
providers:
    app_user_provider:
        entity:
            class: App\Entity\User
            property: email
```

Ici notre provider (que l'on a nommé app_user_provider) est la classe App\Entity\User, cette classe représente les utilisateurs de notre site. La seule raison pour laquelle cela marche est que notre classe est mappé par Doctrine, donc les objets de la classe User représente les données entrées dans la table 'user' de notre base de donnée. Ainsi Symfony va finalement aller dans cette table et comparer le mot de passe crypté enregistré et le mot de passe entré précedemment. Voici la forme sous laquelle cela apparait en base de donnée :

9	Thierry Jacques	$2y$13$nj26u3n0PBFAmAep7euS5.fi1dvg35hMWwfgcj.0l2L...	lucie.dupont@club-internet.fr	["ROLE_USER"]
10	François Gonzalez	$2y$13$Yc5UYRvsqgc5DX2ukWTlde6auZBmNzqp1Z1Y352l7bc...	guillet.nath@thierry.fr	["ROLE_USER"]

Entité User :

Afin qu'une entité soit utilisée comme provider par Symfony il faut respecter certaines conditions. La première est l'implementation de 2 interfaces : UserInterface et PasswordAuthenticatedUserInterface, cela entraine l'implémentation d'un certain nombre de méthodes : getRoles(), eraseCredentials(), getUserIdentifier(), et getPassword().

getUserIdentifier() renvoie le champ qui doit être utilisé pour l'identification. Ici nous avons choisi l'email de l'utilisateur, cela peut être changé mais alors il ne faudra pas oublier de le préciser egalement dans le fichier de configuration security.yalm dans la section providers à l'entrée property (voir juste au-dessus).

getPassword() permet de recuperer le mot de passe et eraseCredentials() peut être laissé vide. getRoles() renvoie le rôle de l'utilisateur, cette methode doit obligatoirement renvoyé un rôle. Par défault, Symfony remplit la methode de cette façon :

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

    return array_unique($roles);
}
```

Nous avons choisit d'attribuer un rôle à l'utilisateur lors de sa création, ainsi nous avons reécrit cette methode. On peut créer autant de rôles que l'on veut mais leur nomenclature doit commencer par 'ROLE_' sinon ils ne seront pas reconnus.

Base de donnée :

Nos champs doivent être mappés pour permettre à la classe User de jouer son rôle de provider. Premierement, le champ choisit pour identifier l'utilisateur doit avoir des valeurs uniques. Ainsi il est necessaire de définir un UniqueEntity sur ce champ :

```php
#[UniqueEntity(fields: ['email'], message: 'Ce champ doit être unique')]
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
```

Ensuite on a mappé nos champ via l'annotation ORM et Assert :
'

```php
    #[ORM\Column(type: 'string', length: 60, unique: true)]
    #[Assert\NotBlank(message: 'Vous devez saisir une adresse email.')]
    #[Assert\Email(message:'Le format de l\'adresse n\'est pas correcte.')]
```

Si vous apportez des modifications sur la classe User ulterieurement, n'oubliez pas de créer une migration ensuite afin que ces modifications apparaissent en base de données egalement , pour cela utiliser les commandes :

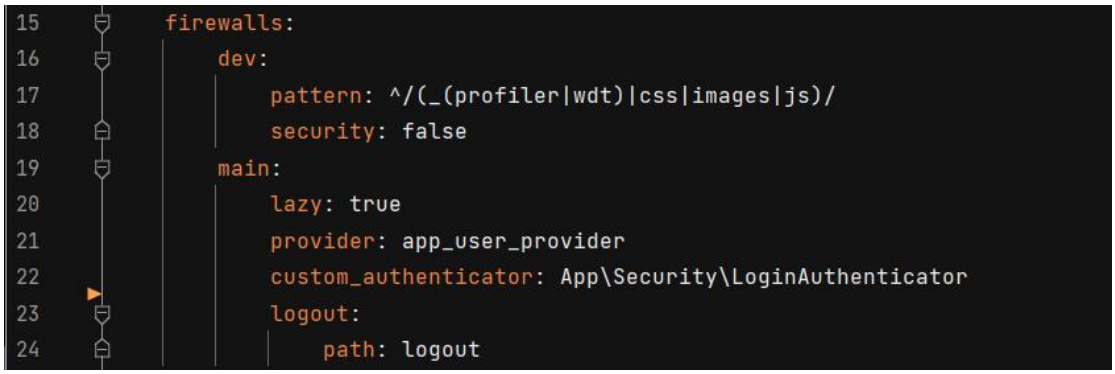
```
Ikan Hiu@Suranadi MINGW64 ~/Desktop/Symfony/P8-ToDoApp (deva)
$ php bin/console make:migration
```

Puis :

```
Ikan Hiu@Suranadi MINGW64 ~/Desktop/Symfony/P8-ToDoApp (deva)
$ php bin/console doctrine:migrations:migrate
```

Firewalls :

Les firewalls sont les pare-feu de l'application. Ils se définissent dans le fichier config/package/security.yaml security: .On voit ici que l'on a deux : le main et le dev.



```yaml
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        lazy: true
        provider: app_user_provider
        custom_authenticator: App\Security\LoginAuthenticator
        logout:
            path: logout
```

Le main est le pare-feu principal, c'est ici que son défini les paramètres principaux de la sécurité en fonction des environnements. Main est le pare-feu utilisé en production, dev est celui utilisé en developpement c'est-à-dire augmenté des outils de déboggage et de profiling. Dans le pare-feu main on voit plusieurs valeurs : lazy signifie que l'on va économiser en ressources, provider c'est le provider que nous avons créer précédemment (ici on se souvient que l'on a un provider que l'on a appelé app_user_provider et qui pointe sur la classe App\Entity\User et sur la propriété email pour les distinguer), custom_authenticator c'est la classe chargée de l'authentification et logout c'est l'url qui permet la déconnexion (ici c'est : /logout).

Access-Control :

Le controle d'accès permet de reserver des parties de notre application en fonction des rôles attribués. Dans notre application nous avons deux

types de rôles : les ROLE_USER et les ROLE_ADMIN.

```
36      access_control:
37          - { path: ^/login, roles: PUBLIC_ACCESS  }
38          - { path: ^/, roles: ROLE_USER }
39          - { path: ^/users, roles: ROLE_ADMIN }
```

Ici, notre application se définit comme tel : la route /login est accessible à tous, l'ensemble des autres routes de l'application (/) demandent à minima d'avoir le ROLE_USER, la partie de gestion des utilisateurs (/users) n'est accessible qu'aux utilisateurs possedant le ROLE_ADMIN.

Role Hierarchy :

Le role hierarchie permet de définir une hierarchie dans les rôles, ici on dit simplement que les ROLE_ADMIN en plus des droits qui leurs sont propres peuvent hérités de tous les droits des ROLE_USER.

```
40      role_hierarchy:
41          ROLE_ADMIN: ROLE_USER
```

Formulaire d'authentification :

Lorsqu'un utilisateur tente de se connecter, il se rend sur l'adresse /login, la route /login est prise en charge par le SecurityController. On voit que la methode loginAction() se contente de retourner une view : login.html.twig :

```
<form action="{{ path('login') }}" method="post" class="text-center p-3" style="...">
    <div class="my-3">
        <label for="username">Entrer votre email :</label>
        <input type="text" id="username" name="_username" value="{{ last_username }}" /> <br/>
    </div>

    <div class="my-3">
        <label for="password">Entrer votre mot de passe :</label>
        <input type="password" id="password" name="_password" />
        <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}" > <br/>
    </div>
    <button class="btn btn-success" type="submit">Se connecter</button> <br/>
</form>
```

Ce formulaire est simple, il s'agit d'un formulaire a deux champs visibles : username et password, et un troisième champ invisible qui génère un token d'authentification.

Soumission du formulaire :

Lorsque l'utilisateur clique sur «Se connecter» alors il est redirigé vers la classe chargée de l'authentification défini précédemment dans le firewall, c'est-à-dire ici dans le App\Security\LoginAuthenticator. Cette classe doit hériter de AbstractLoginFormAuthenticator. Dans cette classe parente, on peut voir la fonction supports() qui va verifier que les conditions sont remplis afin de recourir à l'authentificateur et plus specialement à sa methode authenticate() :

```php
public function authenticate(Request $request): Passport
{
    $email = $request->request->get(key: '_username', default: '');
    $request->getSession()->set(Security::LAST_USERNAME, $email);

    return new Passport(
        new UserBadge($email),
        new PasswordCredentials($request->request->get(key: '_password', default: '')),
        [
            new CsrfTokenBadge(csrfTokenId: 'authenticate', $request->request->get(key: '_csrf_token')),
        ]
    );
```

Sans rentrer dans les détails, cette methode retourne un «passport». Ce passeport prend en argument 3 objets : un objet UserBadge qui va chercher l'utilisateur, un objet PasswordCredentials qui va verifier la validité du mot de passe et un objet CsrfTokenBadge qui va verifier la validité du token.

Gestion de l'authentification :

Une fois la methode authenticate() finie il n'y a que 2 choix possibles : l'authentification a réussie ou elle a échouée. Si elle a échoué alors une methode du AbstractLoginFormAuthenticator est appelée : c'est onAuthenticationFailure(), si l'authentification a réussie alors c'est la methode onAuthenticationSuccess() qui est appelée. Il est possible de personnaliser la redirection souhaitée lors de l'authentification en remplaçant 'homepage' par le nom de la route voulue :

```php
public function onAuthenticationSuccess(Request $request, TokenInterface $token, string $firewal
{
    if ($targetPath = $this->getTargetPath($request->getSession(), $firewallName)) {
        return new RedirectResponse($targetPath);
    }
    return new RedirectResponse($this->urlGenerator->generate( name: 'homepage'));
}
```

A noter que lorsqu'un utilisateur demande une url sécurisé mais qu'il n'est pas identifié c'est la methode start() du AbstractLoginFormAuthenticator qui est appelée.

Suppression et autorisation :

La suppression des tâches est soumises à des regles précises. Pour rappel, seul l'auteur-créateur de la tâche peut la supprimer. Il existe egalement un cas particulier dans le sens où dans la première version de l'application les tâches n'étaient pas rattachées à des utilisateurs, ainsi de nombreuses tâches se retrouvent aujourd'hui avec un auteur «anonyme». Il a été décidé que seul les utilisateurs aux ROLE_ADMIN pourraient supprimer les tâches d'auteurs anonymes. Pour repondre à ces différentes problematiques nous avons créer un Voter. Nous avons activé le Voter grâce à une ligne de code que l'on retrouve dans la methode delete() (qui s'occupe de gérer les appels à l'url /tasks/{id}/delete) du App\Controller\TaskController :
$this->denyAccessUnlessGranted('delete', $task)
Cela signifie que lors de l'appel à cette route, on va verifier les autorisations liées au mot-clé 'delete'.
Si l'on se rend maintenant App\Security\Voter\TaskVoter, on retrouve une methode supports() qui verifie la présence du mot-clé 'delete' et de la tache précedemment appelée via l'url. Si la methode renvoie 'true' alors la methode voteOnAttribute() va être appelée.

```php
protected function supports(string $attribute, $object): bool
{
    if ('delete' !== $attribute) {
        return false;
    }
    if (!$object instanceof Task) {
        return false;
    }
    return true;
}
```

La methode voteOnAttribute() recupère l'utilisateur connecté et renvoie «true» si l'auteur-créateur de la tâche est l'utilisateur connecté ou si la tâche et anonyme et l'utilisateur connecté est un ROLE_ADMIN. Dans tout les autres cas, elle renverra «false». Ainsi le Voter agit comme un petit pare-feu spécialement pour la methode delete() du TaskController.

Conclusion :

Le projet ToDo&Co a été mis à jour et bénéficie maintenant d'une dimension sécuritaire fonctionnelle basée sur l'authentification et l'autorisation . Les erreurs remontées lors de l'audit ont été corrigées et les tests fonctionnels et unitaires implementés.

Pour la suite du projet il convient aux futurs developpeurs de prendre connaissance du depot GitHub et de cloner le projet puis de consulter la documentation (ReadMe, Contributing, Audit, ce document) et de perseverer dans les bonnes pratiques de developpement en suivant les principes SOLID et les regles des PSR.

Afin de garantir la solidité de l'application, à chaque classes et fonctionnalités rajoutées devront être developpés les tests qui lui sont associées, et afin de garantir la lisibilité et la compréhension du code par de futurs developpeurs, la documentation liées aux nouvelles fonctionnalités developpées devra être écrites.