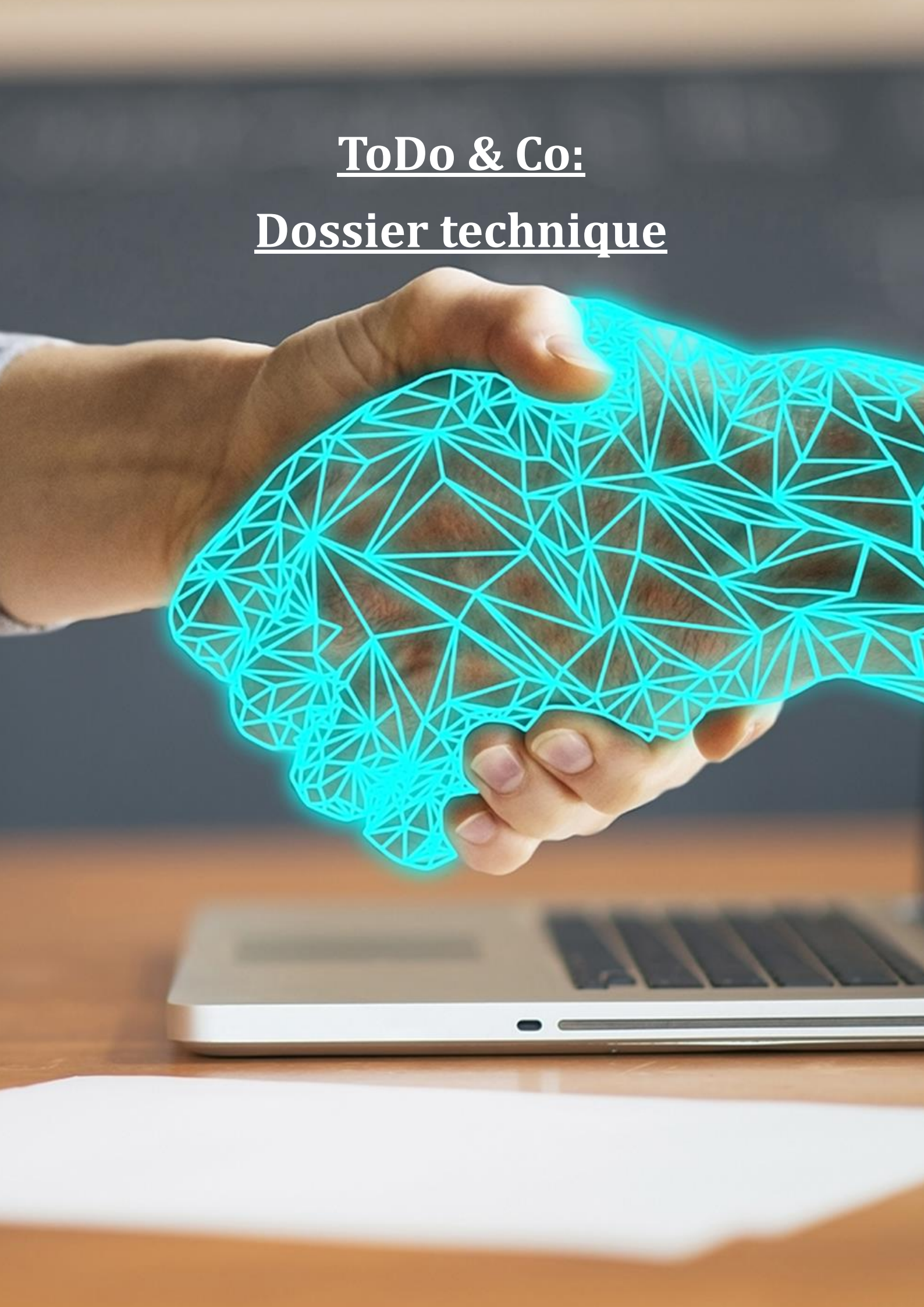
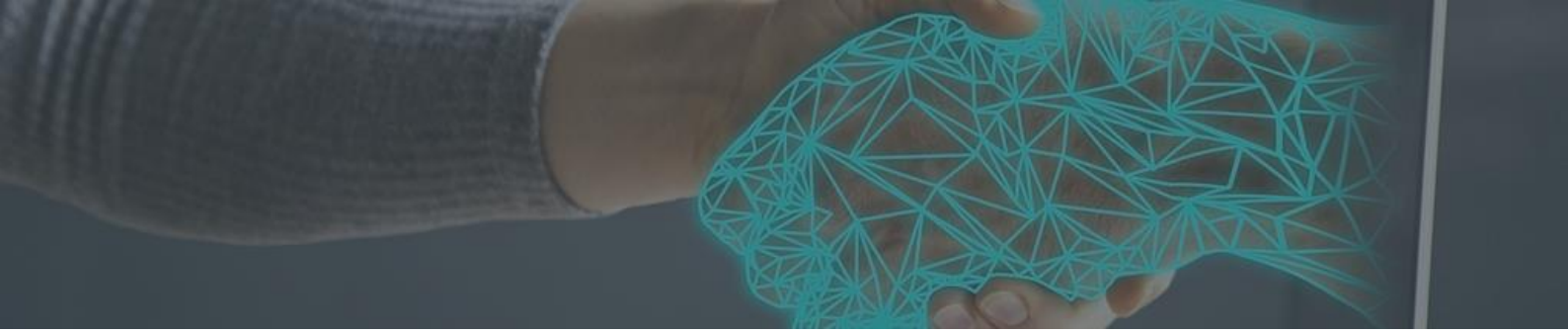
ToDo & Co:

Dossier technique





Introduction:

Contexte:

ToDo & Co est une startup dont le cœur de métier est une application permettant de gérer ses tâches quotidiennes. Le choix du développeur précédent a été d'utiliser le framework PHP Symfony pour créer l'application. L'application a dû être développée à toute vitesse pour permettre de montrer à de potentiels investisseurs que le concept est viable. Par la suite, ToDo & Co a réussi à lever des fonds pour permettre le développement de l'entreprise et surtout de l'application.

Mon rôle ici est donc d'améliorer la qualité de l'application. Pour ce faire, il met demandé de produire une documentation expliquant comment l'implémentation de l'authentification a été faite. Cette documentation se destine aux prochains développeurs juniors qui rejoindront l'équipe. Dans cette documentation, il est possible pour un débutant avec le framework Symfony de :

- Comprendre quel(s) fichier(s) il faut modifier et pourquoi.
- Comment s'opère l'authentification.
- Où sont stockés les utilisateurs.

Le projet :

Le site a été créé sur Symfony 3. Pour avoir un site pouvant être mise à jour, propre et de bonne qualité suivant les Bests Pratices et le principe SOLID, son passage sous Symfony 4 est devenu une obligation. Après avoir réglé les problèmes de dépréciations et celle des librairies utilisés sur le framework. Son passage à Symfony 4.4 a été possible.

Ce document présente seulement la partie authentification, elle ne parle pas des modifications n'ayant pas de raison d'être présenté dans ce document.

Plan:

La mise en place de l'authentification:

a) Installation des packages:

Sa mise en place comme elle est maintenant a été faite après sa mise à jour à Symfony 4.4. Dans un premier temps, il a fallu mettre en place le package security-bundle via composer.

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

Pour créer plus facilement un système d'authentification, j'ai installé maker-bundle via composer

```
composer require symfony/maker-bundle
```

b) L'entité User:

La modification a été faite aussi sur l'entité User. Pourquoi? Car c'est elle qui fait la liaison avec la base de donnée via doctrine (mise à jour précédemment). De plus celle-ci est obligatoire pour utiliser les services de Security par une implémentation de UserInterface. Sur ce site web les modifications on était faite à la main ou via maker avec la commande:

```
php bin/console make:entity
```

*Ce fichier suit le fichier créer par la commande "**php bin/console make:user »***

L'étape suivante a été de créer cette base de données grâce aux entités par des commandes doctrine. *Ne pas oublier de configurer la base de donnée dans le fichier .env et doctrine*.yaml .

```
php bin/console doctrine:database:create
```

```
php bin/console doctrine:schema:update --force
```

c) La configuration:

Pour son fonctionnement, Security doit être configuré. Il existe un fichier de configuration dans le dossier config/packages au nom de security.yaml.

```yaml
security:
    encoders:
        App\Entity\User: bcrypt

    providers:
        app_user_provider:
            entity:
                class: App\Entity\User
                property: username

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        main:
            anonymous: ~
            logout: ~
            guard:
                authenticators:
                    - App\Security\LoginFormAuthenticator

    access_control:
        - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/users, roles: ROLE_ADMIN }
        - { path: ^/, roles: ROLE_USER }
```

Ce fichier comporte plusieurs éléments:

- **encoders:** Permet de contrôler la façon dont les mots de passe son encodé.
- **providers:** Un fournisseur d'utilisateur aidant à recharger des données utilisateur de la session, charger l'utilisateur pour une fonctionnalité,….
- **firewalls:** Définie comment vos utilisateurs pourront s'authentifier
- **access_control:** Permet de protéger les modèles d'URL.

d) Le Controller:

Comme les autres fichiers, le Controller a dû être modifié pour permettre son fonctionnement avec Security. Les modifications ont été faites sur deux éléments. La première, concerne la partie récupérant les erreurs connexion enregistré par Security via la classe AuthenticationUtils . Il permet aussi de récupéré un nom d'utilisateur présent en session. Le deuxième concerne la réponse envoyé à l'utilisateur utilisant la classe Response et Environment, celle-ci permet de suivre le principe SOLID *"Principe de séparation des interfaces"*.

```
class SecurityController
{
    /**
     * @Route("/login", name="login")
     * @param AuthenticationUtils $authenticationUtils
     * @param Environment $twig
     * @return Response
     * @throws \Twig\Error\LoaderError
     * @throws \Twig\Error\RuntimeError
     * @throws \Twig\Error\SyntaxError
     */
    public function login(AuthenticationUtils $authenticationUtils, Environment $twig):
Response
    {
        $error = $authenticationUtils->getLastAuthenticationError();
        // last username entered by the user
        $lastUsername = $authenticationUtils->getLastUsername();

        $render = $twig->render('security/login.html.twig', [
            'last_username' => $lastUsername,
            'error' => $error,
        ]);
        return new Response($render);
    }

    /**
     * @Route("/logout", name="logout")
     */
    public function logoutCheck()
    {
        // This code is never executed.
    }
}
```

e) L'authentification:

L'authentification sur le site web passe par une classe appelée *Guard authenticator*. Celle-ci donne un contrôle complet sur le processus d'authentification.

La création du fichier a été faite grâce à une commande maker en ligne de commande:

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

Celle-ci demande si elle doit créer un formulaire de connexion, pour le site, le formulaire existé déjà la réponse a été non, elle a demandé le nom de la classe d'authentification. Sur ce site le nom du fichier est *LoginFormAuthenticator* situé dans le dossier *Security*.

```
class LoginFormAuthenticator extends AbstractFormLoginAuthenticator
{
    use TargetPathTrait;

    private $entityManager;
    private $urlGenerator;
    private $csrfTokenManager;
    private $passwordEncoder;

    public function __construct(EntityManagerInterface $entityManager,
UrlGeneratorInterface $urlGenerator,
                                CsrfTokenManagerInterface $csrfTokenManager,
UserPasswordEncoderInterface $passwordEncoder)
    {
        $this->entityManager = $entityManager;
        $this->urlGenerator = $urlGenerator;
        $this->csrfTokenManager = $csrfTokenManager;
        $this->passwordEncoder = $passwordEncoder;
    }
```

Le fichier commence par mettre dans les attributs différents services utilisés pendant l'authentification:
entityManager est une interface permettant la liaison avec la base de données.
urlGenerator est un générateur d'url.
csrfTokenManager est un service s'occupant des jetons d'authentifications.
passwordEncoder traite l'encodage et la validité des mots de passe.

```
public function supports(Request $request)
{
    return 'login' === $request->attributes->get('_route')
        && $request->isMethod('POST');
}
```

La première méthode ***supports()*** est appelée à chaque requête. Elle retourne `true` si cette demande contient des informations d'authentification que cet authentificateur sait traiter sinon elle retourne `false`. Lorsque nous soumettons le formulaire de connexion, il s'envoie à `/login`. Ainsi, notre authentificateur doit *seulement* essayer d'authentifier l'utilisateur dans cette situation exacte. En d'autres termes, cela vérifie si l'URL l'est `/login`. Nous voulons également que notre authentificateur essaie de connecter l'utilisateur uniquement s'il s'agit d'une demande POST (`$request->isMethod('POST')`).

```
public function getCredentials(Request $request)
{
    $credentials = [
        'username' => $request->request->get('username'),
        'password' => $request->request->get('password'),
        'csrf_token' => $request->request->get('_csrf_token'),
    ];
    $request->getSession()->set(
        Security::LAST_USERNAME,
        $credentials['username']
    );
    return $credentials;
}
```

getCredentials récupère les informations d'authentification lors de la demande et les renvois dans un tableau. Comme vous pouvez le voire il met aussi en session une variable contenant le nom d'utilisateur.

```php
public function getUser($credentials, UserProviderInterface $userProvider)
{
    $token = new CsrfToken('authenticate', $credentials['csrf_token']);
    if (!$this->csrfTokenManager->isTokenValid($token)) {
        throw new InvalidCsrfTokenException();
    }

    $user = $this->entityManager->getRepository(User::class)->findOneBy(['username' =>
$credentials['username']]);

    if (!$user) {
        // fail authentication with a custom error
        throw new CustomUserMessageAuthenticationException('L\'utilisateur est
introuvable.');
    }

    return $user;
}
```

getUser sert à récupérer un utilisateur dans la base de données, si un utilisateur n'est pas trouvé il soulève une exception. Il vérifie aussi la validité des données par un token d'authentification.

```php
public function checkCredentials($credentials, UserInterface $user)
{
    $correspondingAccount = $this->passwordEncoder->isPasswordValid($user,
$credentials['password']);

    if (!$correspondingAccount) {
        throw new CustomUserMessageAuthenticationException('le mot de passe n\'est pas
bon.');
    }

    return $correspondingAccount;
}
```

checkCredentials permet de vérifier le mot de passe fournie dans le formulaire avec celui de la base de données. Pour ce faire il vérifie le mot de passe avec PasswordEncoder, si celui renvoie false il renvoie une exception .Sinon il renvoie true, l'authentification est réussie.

```php
public function onAuthenticationSuccess(Request $request, TokenInterface $token,
$providerKey)
{
    if ($targetPath = $this->getTargetPath($request->getSession(), $providerKey)) {
        return new RedirectResponse($targetPath);
    }

    return new RedirectResponse($this->urlGenerator->generate('homepage'));
}
```

onAuthenticationSucess est appelé si l'authentification à checkCredentials est un succès (renvoie true). Il permet une redirection de l'utilisateur après s'être connecté. Si l'utilisateur voulez aller sur une page spécifique il sera immédiatement redirigé sur celle-ci sinon il sera redirigé sur la page d'accueil du site.

```
protected function getLoginUrl()
{
    return $this->urlGenerator->generate('login');
}
```

getLoginUrl sert à la redirection de l'utilisateur si l'authentification échoue.

f) Le Template:

```
% block body %}
    <form method="post">
    {% if error %}
        <div class="alert alert-danger">{{ error.messageKey|trans(error.messageData,
'security') }}</div>
    {% endif %}

        <label for="username">Nom d'utilisateur :</label>
        <input type="text" id="username" name="username" value="{{ last_username }}"
               class="form-control" placeholder="username" required/>

        <label for="inputPassword" class="sr-only">Mot de passe</label>
        <input type="password" name="password" id="inputPassword"
               class="form-control" placeholder="Password" required/>

        <input type="hidden" name="_csrf_token"
               value="{{ csrf_token('authenticate') }}"
        >

        <div class="checkbox mb-3">
            <label>
                <input type="checkbox" name="_remember_me"> Se souvenir de moi
            </label>
        </div>

        <button name="button" class="btn btn-lg btn-primary" type="submit">
            Connexion
        </button>
    </form>
{% endblock %}
```

Le Template de Security est un simple formulaire de connexion html utilisant Twig pour la mise en place des variables. Dans celle-ci apparaît:

- Le nom d'utilisateur (username)
- Le mot de passe (password)
- Le token d'authentification (token)
- Le bouton checkbox de rappel des identifiants
- Le bouton de soumission

Son fonctionnement:

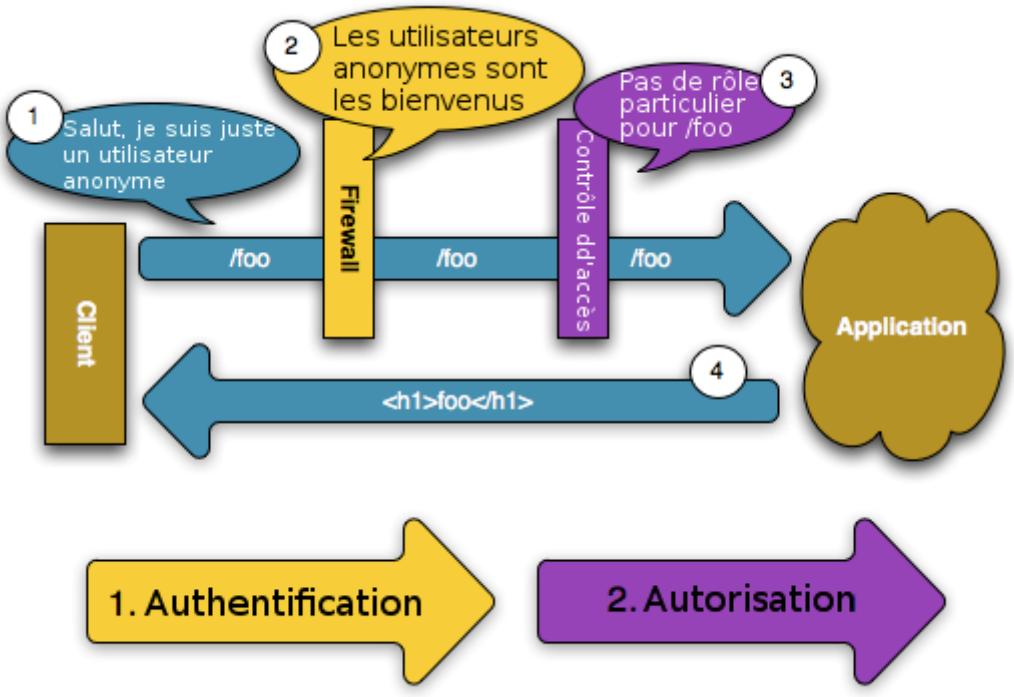
L'authentification sous Symfony est très poussée, pour atteindre ce but, Symfony a bien séparé deux mécanismes différents : l'authentification et l'autorisation. Ce qui gère l'authentification dans Symfony s'appelle un *firewall*. L'authentification est le processus qui va définir qui vous êtes, en tant que visiteur. L'enjeu est vraiment très simple : soit vous ne vous êtes pas identifié sur le site et vous êtes un anonyme, soit vous vous êtes identifié (via le formulaire d'identification ou via un cookie « Se souvenir de moi ») et vous êtes un membre du site. Ainsi vous pourrez sécuriser des parties de votre site Internet juste en forçant le visiteur à être un membre authentifié. Si le visiteur l'est, le firewall va le laisser passer, sinon il le redirigera sur la page d'identification.

L'autorisation est le processus qui va déterminer si vous avez le droit d'accéder à la ressource (la page) demandée. Il agit donc après le firewall. Ce qui gère l'autorisation dans Symfony s'appelle l'*access control*. Par exemple, un membre identifié *lambda* aura accès à la liste de sujets d'un forum, mais ne peut pas supprimer de sujet. Seuls les membres disposant des droits d'administrateur le peuvent, c'est ce que l'access control va vérifier.

Scenarios:

Je suis anonyme, et je veux accéder à la page `/foo` qui ne requiert pas de droits

Dans cet exemple, un visiteur anonyme souhaite accéder à la page `/foo`. Cette page ne requiert pas de droits particuliers, donc tous ceux qui ont réussi à passer le firewall peuvent y avoir accès. La figure suivante montre le processus.

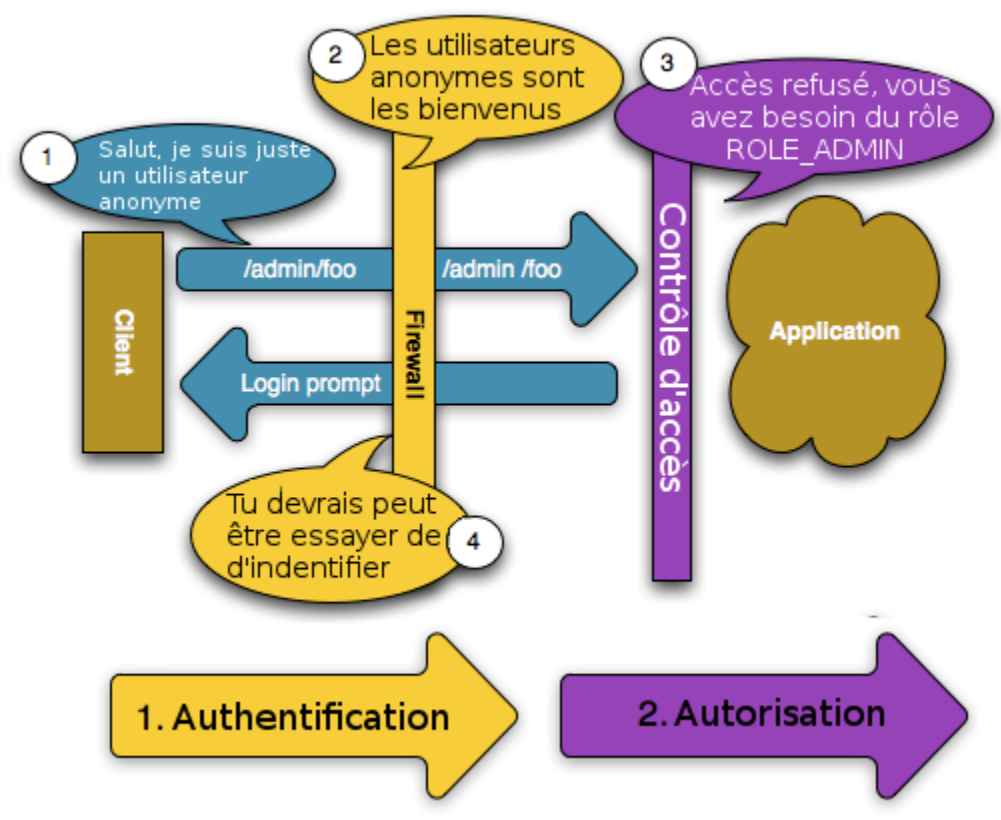


Sur ce schéma, vous distinguez bien le firewall d'un côté et l'access control (contrôle d'accès) de l'autre. Reprenons-le ensemble pour bien comprendre :

1. Le visiteur n'est pas identifié, il est anonyme, et tente d'accéder à la page `/foo`.
2. Le firewall est configuré de telle manière qu'il n'est pas nécessaire d'être identifié pour accéder à la page `/foo`. Il laisse donc passer notre visiteur anonyme.
3. Le contrôle d'accès regarde si la page `/foo` requiert des droits d'accès : il n'y en a pas. Il laisse donc passer notre visiteur, qui n'a aucun droit particulier.
4. Le visiteur a donc accès à la page `/foo`.

Je suis anonyme, et je veux accéder à la page /admin/foo qui requiert certains droits

Dans cet exemple, c'est le même visiteur anonyme qui veut accéder à la page `/admin/foo`. Mais cette fois, la page `/admin/foo` requiert le rôle `ROLE_ADMIN`; c'est un droit particulier, nous le verrons plus loin. Notre visiteur va se faire refuser l'accès à la page, la figure suivante montre comment.

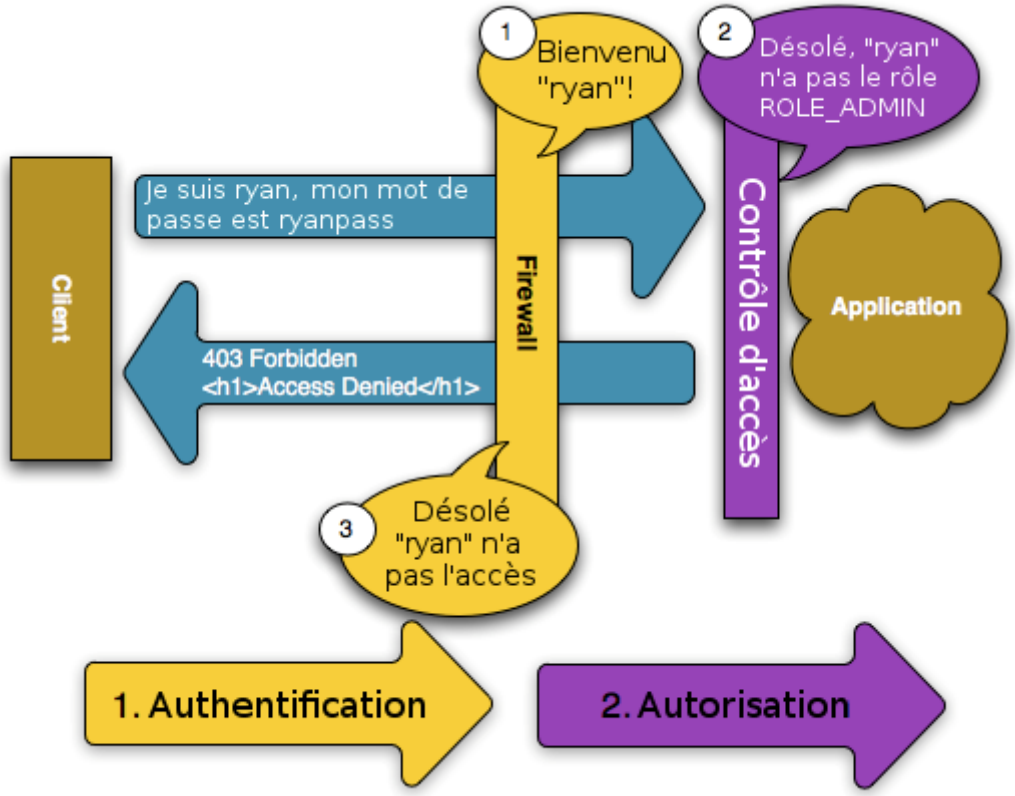


Voici le processus pas à pas :

1. Le visiteur n'est pas identifié, il est toujours anonyme, et tente d'accéder à la page `/admin/foo`.
2. Le firewall est configuré de manière qu'il ne soit pas nécessaire d'être identifié pour accéder à la page `/admin/foo`. Il laisse donc passer notre visiteur.
3. Le contrôle d'accès regarde si la page `/admin/foo` requiert des droits d'accès : oui, il faut le rôle `ROLE_ADMIN`. Le visiteur n'a pas ce rôle, donc le contrôle d'accès lui interdit l'accès à la page `/admin/foo`.
4. Le visiteur n'a donc pas accès à la page `/admin/foo`, et se fait rediriger sur la page d'identification.

Je suis identifié, et je veux accéder à la page/admin/foo qui requiert certains droits

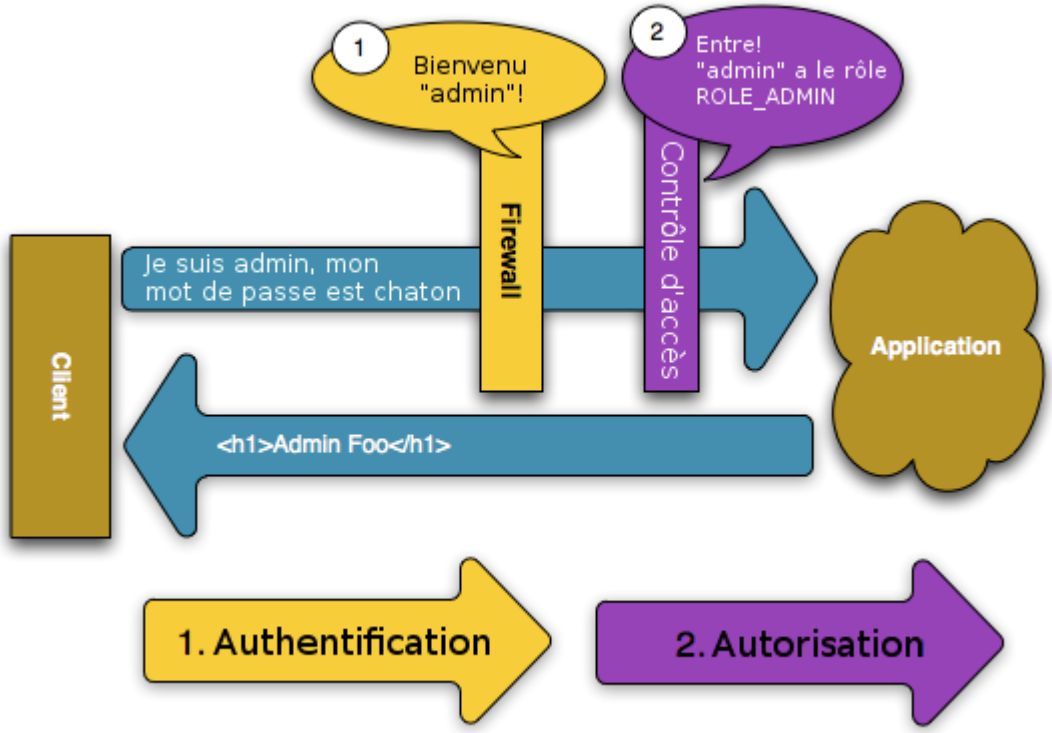
Cet exemple est le même que précédemment, sauf que cette fois notre visiteur est identifié, il s'appelle Ryan. Il n'est donc plus anonyme.



1. Ryan s'identifie et il tente d'accéder à la page `/admin/foo`. D'abord, le firewall confirme l'authentification de Ryan (c'est son rôle !). Visiblement c'est bon, il laisse donc passer Ryan.
2. Le contrôle d'accès regarde si la page `/admin/foo` requiert des droits d'accès : oui, il faut le rôle `ROLE_ADMIN`, que Ryan n'a pas. Il interdit donc l'accès à la page `/admin/foo` à Ryan.
3. Ryan n'a pas accès à la page `/admin/foo` non pas parce qu'il ne s'est pas identifié, mais parce que son compte utilisateur n'a pas les droits suffisants. Le contrôle d'accès lui affiche donc une page d'erreur lui disant qu'il n'a pas les droits suffisants.

Je suis identifié, et je veux accéder à la page /admin/foo qui requiert des droits que j'ai.

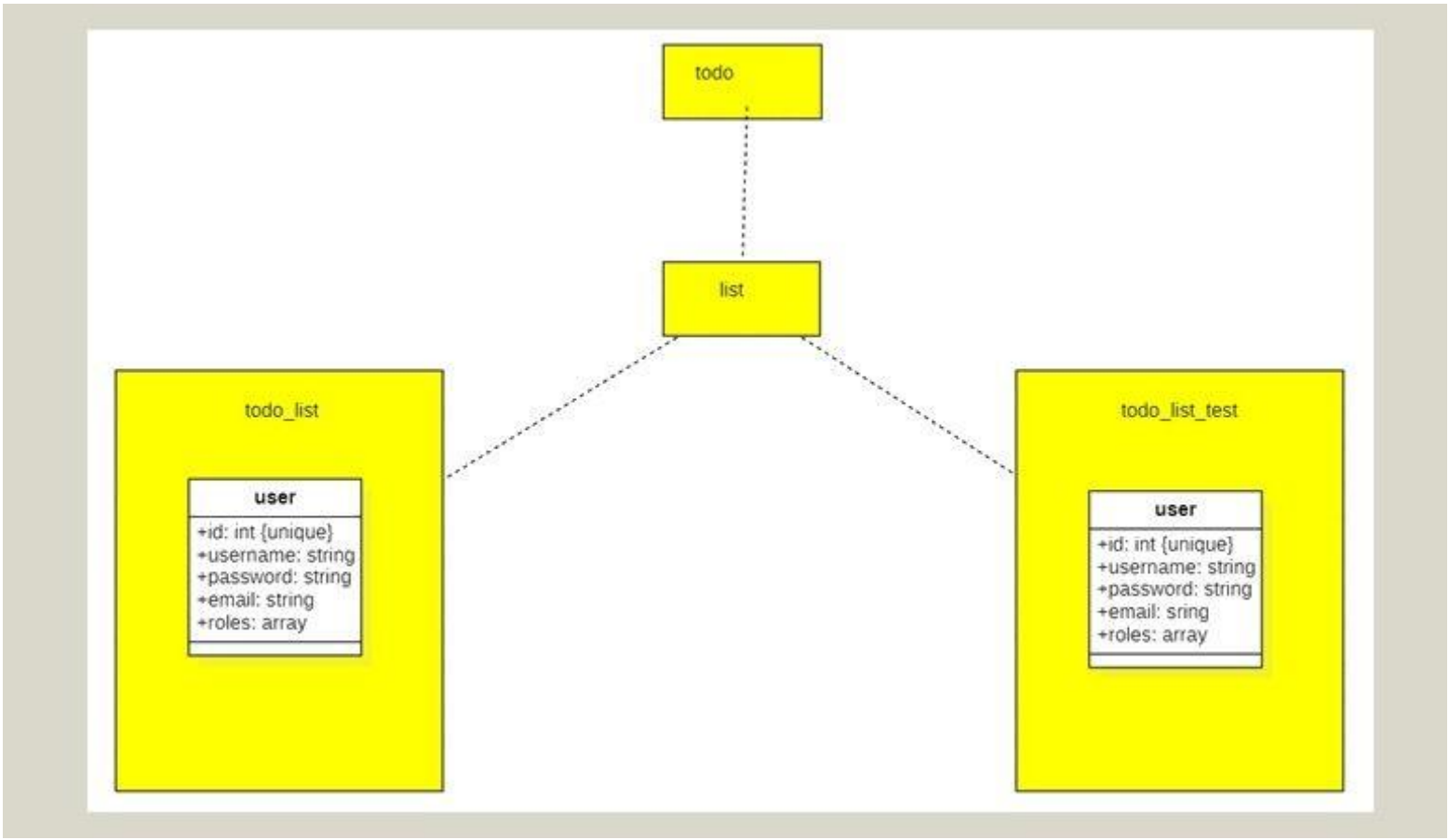
Ici, nous sommes maintenant identifiés en tant qu'administrateur, on a donc le rôle `ROLE_ADMIN`! Du coup, nous pouvons accéder à la page `/admin/foo`, comme le montre la figure suivante.



1. L'utilisateur admin s'identifie, et il tente d'accéder à la page `/admin/foo`. D'abord, le firewall confirme l'authentification d'admin. Ici aussi, c'est bon, il laisse donc passer admin.
2. Le contrôle d'accès regarde si la page `/admin/foo` requiert des droits d'accès : oui, il faut le rôle `ROLE_ADMIN`, qu'admin à bien. Il laisse donc passer l'utilisateur.
3. L'utilisateur admin a alors accès à la page `/admin/foo`, car il est identifié et il dispose des droits nécessaires.

Le stockage des utilisateurs dans la Base de données:

Sur l'application, le stockage des données est faîte avec MySQL et la liaison entre la base de données et l'application et faite avec Doctrine. Les données représentées dans la base de données ressemblent à celle dans les entités.



Sur cet organigramme vous pouvez voir une partie de la base de données concernant les données d'authentification. Celle-ci est divisée en deux sous base identique, une pour le développement, l'autre pour les tests.

Les données utilisées sont :

-un identifiant, unique, en « integer » fait de chiffre, celle-ci est définie automatiquement par MySQL à la création du compte.
-un nom d'utilisateur, unique, en « string » fait de caractère et de chiffre limité à 25 caractères.
-un mot de passe, en « string », encoder suivant la configuration dans le fichier security.yaml, limite 64 caractères.
-un email, unique, « string », limité à 60 caractères.
-un rôle, en json, doit correspondre à un rôle suivant les celles écrite dans la partie « access_control » dans le fichier security.yaml.

Les données sont enregistrées, modifiés ou supprimés en utilisant Doctrine.

a) Création d'un utilisateur:

La création d'un utilisateur est écrite dans le fichier *CreateUserController.php*.

```
$user = new User();
$form = $formFactory->create(UserType::class, $user);

$form->handleRequest($request);
if ($form->isSubmitted()) {
    $password = $passwordEncoder->encodePassword($user, $user->getPassword());
    $user->setPassword($password);

    $em->persist($user);
    $em->flush();
```

Pour créer un compte utilisateur, il faut créer un objet User (entité) correspondant aux données pouvant être enregistré dans la base de données. De celle-ci on fait correspondre les attributs avec celle du formulaire mise en place dans le fichier UserType.php affiché dans le Template create.html.twig. A sa soumission, on récupère un objet User remplie avec les données transmise par l'utilisateur puis on remplies ou modifies les données. Dans ce document, on peut voir qu'après la soumission, le mot de passe est encoder avec PasswordEncoder puis mis dans l'objet User grâce à la méthode setPassword(). L'objet User étant créé, il est mise en place dans la base de données avec les méthodes persist() et flush().

b) Modification d'un utilisateur:

La modification d'un utilisateur est écrite dans le fichier *EditUserController.php*.

```
 $form = $formFactory->create(UserType::class, $user);

$form->handleRequest($request);
if ($form->isSubmitted()) {
    $password = $passwordEncoder->encodePassword($user, $user->getPassword());
    $user->setPassword($password);

    $em->flush();
```

Pour modifier un compte utilisateur, il faut créer un formulaire avec les données de l'utilisateur modifié. Le reste suit le même principe que la création d'un compte utilisateur. Comme vous pouvez le voir il y a pas besoins de mettre la méthode persist() car le compte existe déjà en base de données.

Dans ce site, il n'y a pas de contrôleur permettant de supprimer un utilisateur.

Les modifications possibles sur la partie authentification

Comme vous pouvez le voir, il y a plusieurs niveaux de fichiers pouvant être modifies sans conséquence pour les librairies que ça soit Security ou le Framework Symfony.

Au niveau de la configuration, dans le fichier Security.yaml, vous pouvez modifier ajouter ou supprimer des règles sur le fonctionnement de l'authentification ou la sécurité des pages. Comme écrit plus haut dans ce document, vous pouvez modifier dans celui-ci :

encoders: Changer de type d'encodage est passé par exemple sur argon2i.
providers: Créer une nouvelle entité est définissez celle-ci pour la connexion à l'application.
firewalls: On peut par exemple créer un nouveau pare feu, restreindre le pare feu, …
access_control: On peut créer un nouveau rôle pour certaines pages spécifiques.

Le contrôler peux être modifiés sans conséquence sur le reste des fichiers authentification.

Le Template peut lui aussi être modifié. Attention, le formulaire doit contenir les éléments demandés pour pouvoir être connecté.

Le fichier d'authentification lui peut être modifiés dans toute les méthodes comme expliquer plus haut dans le document, celui-ci a était créé pour cette application. Attention tout de même à modifier suivant les recommandations de ce document, la documentation Symfony et les principes SOLID.

Source Externe :

Pour avoir des explications supplémentaire vous pouvez aller sur le site de Symfony présentant la partie Security. Vous pouvez voir aussi les Best Practices et la libraire Doctrine qui pourrez-vous servir dans certaine situation. Les adresses utiles pour cette partie sont mises en dessous.

https://symfony.com/doc/current/security.html
https://symfony.com/doc/4.4/best_practices.html
https://symfony.com/doc/4.4/doctrine.html
http://blog.ezoqc.com/5-exemples-faciles-pour-comprendre-les-principes-solid/
https://github.com/michaelgtfr/todoList/