COMPOSANT SECURITE DE

Guide du

Table des matières

I) Principe général

1.1) Le processus

Un utilisateur tente d'accéder à une ressource protégée, le processus suivra toujours le même schéma

1. Un utilisateur veut accéder à une ressource protégée
2. Le firewall redirige l'utilisateur au formulaire de connexion
3. L'utilisateur soumet ses informations d'identification (par exemple login et mot de passe)
4. Le firewall authentifie l'utilisateur
5. L'utilisateur authentifié renvoie la requête initiale
6. Le contrôle d'accès vérifie les droits de l'utilisateur, et autorise ou non l'accès à la ressource protégée

Les étapes sont simples, mais très souples. En effet, derrière le mot « authentification » il y a différents procédés : un formulaire de connexion classique, mais également l'authentification via Facebook, Google, etc., ou via les certificats X.509.

1.2) L'authentification

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. C'est ce que la procédure d'authentification va déterminer. Ce qui gère l'authentification dans Symfony s'appelle un firewall, ou un pare-feu en français.

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. Cela se fera donc dans les paramètres du firewall, nous les verrons plus en détail par la suite.

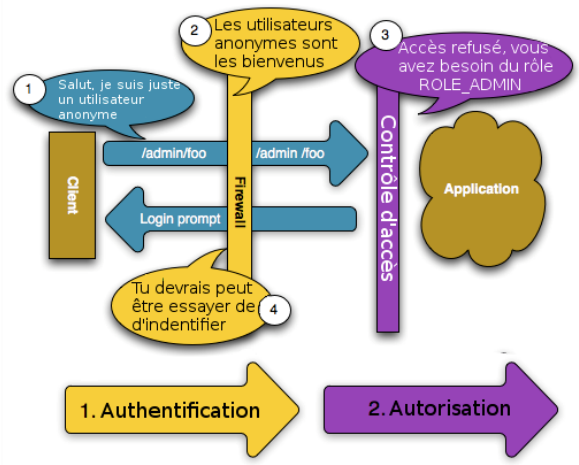
1.3) L'autorisation

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.

1.4) Illustration

Le visiteur est anonyme, et il souhaite accéder à la page/admin/foo qui requiert des droits.

La page/admin/foo requiert le rôle ROLE_ADMIN. Le visiteur va se faire refuser l'accès à la page, la figure suivante montre comment.



Source : [Openclassrooms]

II) Le composant security

2.1) La classe User entity

La classe User doit implémenter obligatoirement la « UserInterface » du composant de sécurité de Symfony. Celle-ci l'oblige à inclure certaines méthodes comme le firewall et le système d'authentification qui seront utilisés pour la vérification de l'utilisateur qui tente de se connecter à votre application.

Les méthodes obligatoires sont les suivantes :

- ✓ getRoles() :

Lorsqu'un utilisateur se connecte, symfony appel cette méthode pour déterminer le rôle de l'utilisateur.

- ✓ getPassword() :

Symfony récupère le mot de passe, si l'application est responsable du stockage et de la vérification des mots de passe de l'utilisateurs.

- ✓ getSalt(),

Symfony renvoie le « salt » utilisé à l'origine pour coder le mot de passe.

- ✓ getUsername(),

Symfony renvoie un identifiant « visuel » pour l'utilisateur.

- ✓ eraseCredentials().

Symfony supprime les données sensibles de l'utilisateur.

Ceci est important si, à un moment donné, des informations sensibles par exemple si le mot de passe est en texte brut et stocké sur cet objet.

Source : Documentation de Symfony

Github « UserInterface »

2.2) Détail du composant security

2.2.1) L'encodage du mot de passe

L'encodage permet de « hasher » le mot de passe afin de le sécuriser. Pour configurer le hashage on configure le fichier security.yaml situé dans le dossier config/packages/. La bonne pratique recommandée par Symfony est de mettre la valeur algorithm : auto afin que le système choisisse le meilleur algorithm en fonction des performances et de la configuration du serveur.

```
encoders:
    App\Entity\User:
        algorithm: auto
```

Figure 1 : Le fichier security.yaml de notre application

Pour l'utiliser on peut prendre la « UserPasswordEncoderInterface » dans les contrôleurs ou handler ou fixtures, pour encoder le mot passe avant son insertion en base de données.

```
$user->setPassword($this->passwordEncoder->encodePassword($user, plainPassword: 'password'));

$manager->persist($user);

$manager->flush();
```

Figure 2 : Exemple d'utilisation lors de la création des fixtures User

Source : Documentation de Symfony

2.2.2) Le provider

Le provider indique à Symfony où se situe l'utilisateur. Il en existe plusieurs types. Pour l'indiquer à Symfony ou aller chercher les utilisateurs on l'implémente dans le fichier security.yaml dans le dossier config/packages/.

Notre application utilise une base de données, nous lui indiquons d'aller chercher le l'utilisateur dans l'entité User :

```yaml
providers:
    from_database:
        entity:
            class: App\Entity\User
            property: username
```

Figure 3 : Provider de l'application

Un autre exemple et l'utilisation d'un provider situé en mémoire « in memory ». Symfony ne recommande pas d'utiliser ce fournisseur dans des applications réelles en raison de ses limites et de la difficulté de gérer les utilisateurs. Mais il peut être implémenté dans les prototypes d'applications et pour les applications limitées qui ne stockent pas les utilisateurs dans des bases de données.

```yaml
providers:
    backend_users:
        memory:
            users:
                john_admin: { password: '$2y$13$jxGxc ... IuqDju', roles: ['ROLE_ADMIN'] }
                jane_admin: { password: '$2y$13$PFi1I ... rGwXCZ', roles: ['ROLE_ADMIN', '
```

Figure 4 : Le provider « in memory » documentation de symfony

Source : Documentation de Symfony

2.2.3) Le firewall

Le firewall indique à Symfony la manière dont s'authentifie les utilisateurs. Un utilisateur peut se connecter par un formulaire classique d'une page web, ou par une API token.

```yaml
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        anonymous: ~
        guard:
            authenticators:
                - App\Security\LoginFormAuthenticator
        logout:
            path: /logout
            target: /
        # activate different ways to authenticate
        # https://symfony.com/doc/current/security.html#firewalls-authentication

        # https://symfony.com/doc/current/security/impersonating_user.html
        # switch_user: true
```

Figure 5 : security.yaml 'implémentation du firewall de l'application

La route de connexion /login dans notre application se fera dans la classe « LoginFormAutehticator » qui est décrite dans la guard ci-dessous.

On indique la route à surveiller /logout pour la déconnexion. La connexion.

Le service Guard de Symfony ajoute une couche de sécurité supplémentaire en vérifiant en base de données : le nom d'utilisateur, l'encodage du mot de passe et le CSRF Token (protection contre les attaques d'injection de code) envoyé par le formulaire de connexion.

Dans notre application c'est la classe « LoginFormAuthenticator » qui génère la route /login pour la connexion.

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

Figure 6 : LoginFormAuthenticator fonction qui génère la route /login

Source : Documentation de Symfony form_login Authentication Provider

Documentation de Symfony Sécurité

2.2.4) Rôle hiérarchie

Le paramètre est implémenté dans le fichier sécurity.yaml. Il permet d'appliquer un héritage au niveau des rôles des utilisateurs.

```yaml
role_hierarchy:
    ROLE_ADMIN: ROLE_USER
```

Figure 6 : Rôle hiérarchie de notre application

Dans notre application l'administrateur aura les mêmes droits que l'utilisateur USER.

Grace au Rôle hiérarchie on peut implémenter plusieurs niveaux d'autorisation le super administrateur ROLE_SUPER_ADMIN héritera du rôle de l'administrateur ROLE_ADMIN qui héritera du rôle de l'utilisateur ROLE_USER.

De ce fait l'utilisateur ROLE_USER bénéficiera du rôle ayant le plus bas niveau d'autorisation.

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

    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

Figure 7 : Implémentation de rôle hiérarchie dans la documentation de Symfony

Source : [Documentation de Symfony Sécurité]

2.2.5) Access control

Le contrôle d'accès nous permet de définir des droits sur les routes. Cela nous permet d'indiquer quel utilisateur aura accès à tel ou tel route.

Vous pouvez donc facilement interdire l'accès à une route si l'utilisateur n'a pas le niveau suffisant d'autorisation.

L'exemple ci-dessous montre que pour accéder à /login (page d'authentification) l'utilisateur n'a pas besoin d'avoir un rôle particulier IS_AUTHENTICATED_ANONYMOUSLY contrairement aux routes /tasks, /users, /delete ou un rôle est obligatoire pour y accéder donc l'utilisateur doit être authentifié et autorisé par le système de sécurité de Symfony.

```yaml
access_control:
    # - { path: ^/admin, roles: ROLE_ADMIN }
    # - { path: ^/profile, roles: ROLE_USER }
    - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY , methods: [GET, POST]}
    - { path: ^/users/create, roles: IS_AUTHENTICATED_ANONYMOUSLY, methods: [GET, POST]}
    - { path: ^/tasks, roles: ROLE_USER, methods: [GET, POST]}
    - { path: ^/users, roles: ROLE_ADMIN, methods: [GET, POST] }
    - { path: ^/user, roles: ROLE_ADMIN, methods: [GET, POST] }
    - { path: ^/delete, roles: ROLE_ADMIN, methods: [GET]}
    - { path: /tasks-Is-Done, roles: ROLE_USER, methods: [GET]}
    - { path: ^/, roles: ROLE_USER, methods: [GET] }
```

Figure 8 : Implémentation du contrôle d'accès de notre application

Source : [Documentation de Symfony Security access_control]

2.2.6) Le voter

Le voter permet de réaliser une action sur un objet. Pour mettre en place un système de Voter il faut tout d'abord créer une classe « TaskVoter » situé dans le dossier App/Voter/. Cette classe permet de définir la logique à exécuter sur une action précise sur un objet.

```php
if ($this->authorization->isGranted( attributes: TaskVoter::EDIT, $task) === true)
```

Figure 9 : Vérification si une tâche peut être édité par l'utilisateur actuel de l'application

La classe « TaskVoter » à une méthode « voteOnAttribute » qui vérifie si l'utilisateur est bien une instance de la classe User et que l'User est bien le propriétaire de la tâche. Si l'utilisateur est autorisé, la méthode « canEdit » pour mettre à jour la tâche ou la méthode « canDelete » est appelée pour supprimer la tâche.

```php
protected function voteOnAttribute($attribute, $subject, TokenInterface $token)
{
    $user = $token->getUser();

    if (!$user instanceof User) {
        return false;
    }

    $task = $subject;

    switch ($attribute) {
        case self::EDIT:
            return $this->canEdit($task, $user);
        case self::DELETE:
            return $this->canDelete($task, $user);
    }

    throw new LogicException( message: 'This code should not be reached!');
}
```

Figure 10 : La méthode « voteOnAttribute » du voter de notre application

```php
private function canEdit(Task $task,User $user): bool
{
    return $user === $task->getUser();
}
```

Figure 11 : Méthode « canEdit » vérifie si l'utilisateur connecté est bien le propriétaire de la tâche.

```php
private function canDelete(Task $task, User $user)
{
    if ($task->getUser() === null & $user->getRoles() === ['ROLE_ADMIN'] or $user === $task->getUser()) {
        return true;
    }
    return false;
}
```

Figure 12 Méthode « canDelete » Vérifie si la tâche n'a pas de propriétaire « anonyme » et que l'utilisateur connecté à un ROLE_ADMIN pour autorise la suppression de la tâche. Et vérifie que si l'utilisateur connecté est le propriétaire de la tâche il pourra la supprimer.

Source : Documentation de Symfony

III) Mise en place de la sécurité dans une application Symfony

3.1) Implémentation de la sécurité

Il y a deux méthodes pour implémenter la sécurité dans une application Symfony.

La première et d'utiliser la « MakerBundle » en ligne de commande qui crée automatique les fichiers nécessaires :

```
D:\Users\Jeromes2\Desktop\Developpement\Todoandco>php ./bin/console make:auth

 What style of authentication do you want? [Empty authenticator]:
  [0] Empty authenticator
  [1] Login form authenticator
 >
```

Figure 13 : Commande « make :auth » dans le terminal créant les fichiers nécessaires

Les fichiers créés et ou mise à jour sont :

- LoginFormAuthenticator
- Security.yaml
- SecurityController
- Login.html.twig

La deuxième méthode est de créer les fichiers manuellement.

Source : Documentation de Symfony Comment créer un formulaire de connexion

3.2) Utilisation de la sécurité

Pour utiliser la sécurité on fait appel au composant de sécurité dans le code de votre application.

Par exemple dans les contrôleurs ou handler pour interdire ou donner l'accès à certaines fonctions suivant le rôle de l'utilisateur.

Dans les vues il est également possible d'afficher ou de masquer des informations en fonction des rôles par exemple un formulaire, une barre de navigation ou même du texte suivant le rôle de l'utilisateur actuellement connecter.

```
{% if is_granted("ROLE_ADMIN")%}
    <li class="nav-item dropdown">
        <div class="dropdown-menu" aria-labelledby="navbarDropdownMenuLink">
            <a class="dropdown-item" href="/"><i class="fa fa-home" aria-hidden="true"></i> Accueil</a>
            <a class="dropdown-item" href="{{ path('/users/list') }}">
                <i class="fa fa-user" aria-hidden="true"></i>
                Gérer les utilisateurs
            </a>
        </div>
    </li>
{% else %}
    <li class="nav-item">
        <a style="..." class="nav-link" href="">Créer un utilisateur</a>
    </li>
{% endif %}
```

Figure 14 : Menu de navigation de l'application qui affiche si l'utilisateur actuellement connecté a le rôle_admin pourra afficher le lien vers la page « Gérer les utilisateurs »

Dans un contrôleur nous aurons besoin de récupérer l'utilisateur actuellement connecté. Pour ce faire vous pouvez utiliser là l'interface « TaskVoter » implémenté préalablement pour vérifier si l'utilisateur connecté peut supprimer la tâche..

```
if ($this->authorization->isGranted( attributes: TaskVoter::DELETE, $task) === true)
```

Figure 15 : deleteTaskController de notre application

3.2) Utilisation de la sécurité (suite)

On peut utiliser les annotations du bundle « SensioFrameworkExtraBundle » pour sécuriser les contrôleurs.

```
// src/Controller/AdminController.php
// ...

+ use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;

+ /**
+  * Require ROLE_ADMIN for *every* controller method in this class.
+  *
+  * @IsGranted("ROLE_ADMIN")
+  */
class AdminController extends AbstractController
{
+     /**
+      * Require ROLE_ADMIN for only this controller method.
+      *
+      * @IsGranted("ROLE_ADMIN")
+      */
    public function adminDashboard()
    {
        // ...
    }
}
```

Figure 16 : Sécurisation d'un contrôleur à l'aide des annotations

Source : Documentation de Symfony sécurité