Document Technique : Sécurité de l'Application Symfony

Introduction

Ce document technique vise à expliquer l'implémentation de la sécurité dans notre application Symfony, en mettant particulièrement l'accent sur le processus d'authentification. Nous détaillerons les différents éléments impliqués dans ce processus, notamment la classe User, le fichier de configuration security.yaml, le SecurityController et les Voters.

1. Classe User

La classe User représente les utilisateurs de notre application. Elle est responsable de la gestion des informations d'identification et des rôles des utilisateurs. Voici les différents attributs définis par notre entité.

```php
#[ORM\Id]
#[ORM\GeneratedValue]
#[ORM\Column]
private ?int $id = null;

#[ORM\Column(length: 180, unique: true)]
#[Assert\NotBlank]
#[Assert\Email]
private ?string $email = null;

#[ORM\Column]
private array $roles = [];

#[ORM\Column]
#[Assert\NotBlank]
#[Assert\Length(min: 8)]
#[
    Assert\PasswordStrength([
        'minScore'   => PasswordStrength::STRENGTH_MEDIUM,
        'message'    => 'Password.Tooweak',
    ])
]
private ?string $password = null;
```

On notera l’utilisation des Contraints Validator de symfony pour vérifier certains points sensibles comme la validité de l’adresse mail, la longueur et la complexité du mot de passe. La complexité moyenne est un score d’entropie calculé comme nombre de librairie le font, il y a 4 niveau sélectionnable.

Au niveau de l'entité on remarquera que la variable password ne subit pas de transformation particulière car symfony utilise directement l'interface d'authentification qui elle gère la vérification avec la valeur hashé du mot de passe en base de données.

```php
/**
 * 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);
}
```

Create an array
array([mixed $...]): array

Enfin deux paramètres de base sont définis, la variable permettant l'identification unique de l'utilisateur ici l'email et la valeur par défaut du rôle ici 'ROLE_USER'.

2. Fichier de Configuration security.yaml

Le fichier de configuration security.yaml définit la stratégie de sécurité de l'application. Il spécifie les différents pares-feux, les points d'entrée d'authentification, les fournisseurs d'utilisateurs et autres options de configuration de sécurité. Voici un exemple simplifié de ce fichier

```yaml
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\User
                property: email
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: app_user_provider
            custom_authenticator: App\Security\Authenticator
            logout:
                path: app_logout
                # where to redirect after logout
                # target: app_any_route
```

La ligne password_hasher spécifie que l'application choisira automatiquement le hasher de mot de passe implémenter via PasswordAuthenticatedUserInterface, actuellement pour notre version Bcrypt.

La partie providers définit la classe qui va nourrir les informations des utilisateurs, ici l'entité User de notre application, on spécifie également son identifiant unique ici l'email.

La section de configuration du pare-feu prévoit le cas de l'environnement de développement (dev:) de l'application qui autorise l'accès au profiler de symfony, aux outils de debug et l'environnement général de l'application (main:) qui elle renvoie vers les chemins d'authentification et de déconnexion de l'application.

La section access_control est faite pour déterminer l'accès à certains URL pour certains rôles uniquement. Pour notre application il n'est pas nécessaire mais on pourrait imaginer qu'une console back-office est URL spécifié ici uniquement pour les administrateurs.

Enfin la dernière partie concerne l'environnement de test.

```
when@test:
    security:
        password_hashers:

Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterf
ace:
                algorithm: auto
                cost: 4
                time_cost: 3
                memory_cost: 10
```

Elle permet une optimisation des temps de calcul lié à la sécurité.

3. SecurityController

Le SecurityController gère les actions liées à l'authentification, telles que la connexion et la déconnexion des utilisateurs. Voici un exemple de méthodes de ce contrôleur :

```php
#[Route(path: '/login', name: 'app_login')]
public function login(AuthenticationUtils $authenticationUtils): Response
{
    if ($this->getUser()) {
        return $this->redirectToRoute('app_home');
    }

    // get the login error if there is one
    $error = $authenticationUtils->getLastAuthenticationError();
    // last username entered by the user
    $lastUsername = $authenticationUtils->getLastUsername();

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

/**
 * @codeCoverageIgnore
 */
#[Route(path: '/logout', name: 'app_logout')]
public function logout(TranslatorInterface $translator): void
{
    throw new \LogicException('This method can be blank - it will be intercepted by the logout key on
    your firewall.');
}
```

Ce Controller permet de gérer les fonctionnalités de connexion et de déconnexion. Elle permet l'affichage du formulaire de connexion.
En cas d'échec de connexion on peut spécifier la redirection dans la condition if ($this->getUser()). Ici nous avons indiqué la route app_home.

La fonction logout est ici présente pour ne pas déclencher d'erreur de route dans le reste du code symfony. En réalité la configuration du logout est déjà présente de le fichier security.yaml via le firewall.

4. Voters

Les Voters sont des classes qui déterminent si un utilisateur est autorisé à effectuer une action spécifique dans l'application. Ils permettent une granularité fine dans la gestion des autorisations. Voici un exemple de voter :

```php
class TaskVoter extends Voter
    protected function voteOnAttribute(string $attribute, mixed $subject, TokenInterface $token): bool
    {
        $user = $token->getUser();
        // if the user is anonymous, do not grant access
        if (!$user instanceof UserInterface) {
            return false;
        }

        // ... (check conditions and return true to grant permission) ...
        switch ($attribute) {
            case self::EDIT:
                return $this->isAuthorOrAdmin($subject, $user);
                // logic to determine if the user can EDIT
                // return true or false
                break;
            case self::VIEW:
                return true;
                break;
            case self::DELETE:
                return $this->isAuthorOrAdmin($subject, $user);
                break;
            case self::CREATE:
                return true;
                break;
        }

        return false;
    }
```

Le voter va vérifier que l'utilisateur a bien le droit d'effectuer chaque action définit (EDIT, VIEW, DELETE, CREATE). Avant toute chose il convient de vérifier s'il y a bien un utilisateur de connecté, puis ensuite on ajoute notre propre logique ici isAuthoOrAdmin().

```php
private function isAuthorOrAdmin(mixed $subject, UserInterface $user): bool
{
    if ($subject->getAuthor()->getUsername() === 'Anonyme' && in_array('ROLE_ADMIN',$user->getRoles())) {
        return true;
    }
    if ($subject->getAuthor() === $user) {
        return true;
    }
    return false;
}
```

@param mixed $subject

D'un côté on vérifie qu l'utilisateur est bien l'auteur de la tâche en question et de l'autre on vérifie le cas spécifique des tâches anonymes qui elles seront uniquement gérés par des administrateurs.

Enfin on fait intervenir le Voter de la façon suivante dans le Controller approprié :

```php
#[Route('/tasks/create', name: "task_create")]
public function createAction(Request $request, EntityManagerInterface $entityManager, Security $security,
{
    $task = new Task();
    if (!$this->isGranted('TASK_CREATE', $task)) {
        $this->addFlash('error', $translator->trans('Task.Create.Error', [], 'messages'));
        return $this->redirectToRoute('task_list_undone');
    }
    $form = $this->createForm(FormTaskType::class, $task);
    $form->handleRequest($request);
```

La fonction isGranted va vérifier que le sujet task est bien dans le périmètre d'activabilité du Voter en adéquation avec l'utilisateur connecté. Si ce n'est pas le cas ici l'action de création n'aura pas lieu et l'utilisateur sera redirigé.

De la même façon la gestion des utilisateurs réservé aux administrateurs va être vérifier par Voter. On aura une vérification pour la visibilité des utilisateurs et pour le changement de rôle.

```php
protected function voteOnAttribute(string $attribute, mixed $subject, TokenInterface $token): bool
{
    $user = $token->getUser();
    // if the user is anonymous, do not grant access
    if (!$user instanceof UserInterface) {
        return false;
    }

    // ... (check conditions and return true to grant permission) ...
    switch ($attribute) {
        case self::EDIT:
            // logic to determine if the user can EDIT
            return $this->isAdmin($subject, $user);
            break;
        case self::VIEW:
            // logic to determine if the user can VIEW
            return $this->isAdmin($subject, $user);
            break;
    }

    return false;
}
```

Dans le Controller la visibilité sera vérifiée de la façon suivante :

```php
#[Route('/users', name: 'user_list')]
public function listUser( UserRepository $userRepository, Security $security, TranslatorInterface
$translator): Response
{

    if (!$this->isGranted('USER_VIEW', $security->getUser())){
        $this->addFlash('error', $translator->trans('User.View.Error', [], 'messages'));
        return $this->redirectToRoute('app_home');
    }
    $userList = $userRepository->findAll();
```

Conclusion :

En combinant ces différents éléments, notre application Symfony assure une sécurité robuste et efficace. La classe User gère les informations d'identification des utilisateurs, le fichier de configuration security.yaml définit la stratégie de sécurité, le SecurityController gère les actions liées à l'authentification, et les Voters permettent de contrôler finement les autorisations des utilisateurs