Prise en main du

Symfony Security Component

Vincent Lescot, décembre 2018

Openclassrooms projet n°8 - Améliorer une application existante

Sommaire

1. Avant-propos

Le composant sécurité de Symfony est puissant mais peut sembler complexe aux premiers abords. Cette documentation a pour but d'être simple et de fournir suffisamment de détails pour une prise en main du composant.

2. Vocabulaire

Deux définitions élémentaires sont à comprendre pour mieux appréhender la suite.

1) Authentification

L'authentification va définir qui est le visiteur. Par défault, Symfony identifie tout utilisateur comme anonyme. Le système d'authentification, par exemple via un login et mot de passe, changera le statut du visiteur comme étant un membre identifié.

En interne, c'est le firewall (pare-feu) qui s'occupera de l'authentification et de laisser passer l'utilisateur vers l'access control (contrôle des accès), qui lui gère les autorisations.

2) Autorisation

Dans une application, plusieurs accès peuvent être alloués. Par exemple, restreindre l'accès à certaines parties du site, ou bien, pour accorder un droit à la lecture mais pas à la création, la modification ou la suppression.

Allouer ou refuser un accès est géré par l'access control. Son but est de vérifier si un utilisateur authentifié posséde bien le statut demandé par l'application pour accéder à la partie du site demandée, ces statuts sont appelés des Roles.

Si l'accès est autorisé, l'access control va laisser le code s'exécuter dans l'application, plus précisément vers les actions. Sinon, le code redirigera directement l'utilisateur vers un formulaire d'authentification par exemple.

3. Vue d'ensemble

Deux grandes parties du composant sécurité ont déjà été explicitées, le firewall et l'access control, mais il reste « l'encoder », le « provider ».

Dans un système d'authentification, le cryptage des informations est une composante essentielle. L'encoder va déterminer quelles données sont cryptées et avec quel(s) algorithme(s).

Le provider quand à lui va définir comment le firewall doit aller chercher l'utilisateur. Est-ce dans la base de donnée ? En session ? via un token ? Symfony est flexible et permet de définir plusieurs providers à utiliser selon l'accès au site demandé.

Nous savons donc que la sécurité à besoin d'accéder aux données de l'utilisateur via les providers, de cryptés des informations sensibles via les encoders, de déterminer qui est l'utilisateur via le firewall et enfin de laisser passer l'utilisateur via l'access control.

4. Providers

Symfony propose plusieurs providers pour répondre aux différents besoins. Plusieurs providers peuvent être utilisés dans la même application comme nous allons le voir.

1) Entité User : base de données

Pour que Symfony recherche l'utilisateur dans la base de donnée, deux interfaces doivent être implémentés et la partie provider du fichier de configuration doit être remplie comme ci-dessous.

Implementer les interfaces

Pour utiliser le User Provider, deux interfaces doivent être implementées :

- UserInterface sur l'entité. Cette interface passe un contrat pour les méthodes getRoles, getPassword, getSalt, getUsername, eraseCredentials.[1]
- UserLoaderInterface sur le repository de l'entité ciblée. Cette interface a un contrat pour la méthode loadUserByUsername.[2]

Le fichier de configuration

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

providers:
# ce nom est libre et sera référencé au firewall
our_db_provider:
        entity:
# namespace de l'entité ciblée
class: App\Entity\User
# la propriété à chercher par exemple, username, email, etc
property: username
# pour utiliser plusieurs entity managers (voir documentation)
            # manager_name: customer
```

2) Memory

Le provider par memory va définir un utilisateur à l'intérieur même du fichier de configuration. Lors de l'authentification, le firewall regardera simplement les données présentes dans le security.yaml.

Le fichier de configuration

```
providers:
   some_provider_key:
      memory:
         users:
            employee_name:  { password: employee's_password,  roles: ROLE_EMPLOYEE }
            boss_name:      { password: boss's_password,      roles: [ROLE_ADMIN, ROLE_MANAGER ] }
```

[1]Comment implémenter UserInterface - https://symfonycasts.com/screencast/symfony3-security/user-class
[2]Comment implémenter UserLoaderInterface https://symfony.com/doc/current/security/entity_provider.html#using-a-custom-query-to-load-the-user

3) Chain

Le chain provider permet aux firewalls d'utiliser plusieurs providers existant pour rechercher l'utilisateur.

Le fichier de configuration

```
providers:
    chain_provider:
        chain:
            providers: [our_db_provider, some_provider_key]
```

4) Ldap

LDAP (Lightweight Directory Access Protocol) est un protocole permettant d'interroger des annuaires. Cette note[3] vers la documentation de Symfony explique comment aller chercher des utilisateurs via LDAP.

5. Encoder

Le fichier de configuration

```
# config/packages/security.yaml
security:
    encoders:
# use your user class name here
App\Entity\User:
# bcrypt or argon2i are recommended
            # argon2i is more secure, but requires PHP 7.2 or the Sodium extension
algorithm: bcrypt
```

Le firewall encodera alors le mot de passe fournit à l'authentification pour le comparer avec celui préalablement encodé avant l'enregistrement dans la base de données par exemple. L'exemple suivant montre comment procéder.

Encoder le password dans un service

```
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;

class UserBuilder
{
private $passwordEncoder;

public function __construct(UserPasswordEncoderInterface $passwordEncoder)
    {
$this->passwordEncoder = $passwordEncoder;
}

public function createUser()
    {
$user = new User();

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

[3] Authentication et LDAP - https://symfony.com/doc/current/security/ldap.html

6. Authentification et Firewalls

Comme dit précédemment, le firewall va authentifier un utilisateur et décider de le laisser passer ou non vers l'access control.

De nombreuses options sont applicables aux firewalls. Je recommande de prendre le temps de lire la documentation.[4]

Je vais donner quelques exemples d'utilisations.

1) Via formulaire

Fichier de configuration

```yaml
# config/packages/security.yaml
security:
# ...
firewalls:
        main:  # Ce nom n'as pas d'importance pour le reste de la configuration
anomymous: ~
pattern:  ^/
provider: user_provider
form_login:
                login_path: login
check_path: login_check
default_target_path: /
logout: ~
```

La clé *anonymous* précise que le firewall s'applique aux utilisateurs authentifié comme «anon. ». S'il est authentifié par un ROLE spécifique, le firewall ne s'appliquera pas et le laissera passer. Cela évite une authentification à chaque requête.

La clé *pattern*réfère aux URLs gérés par le firewall « main », ici toutes les url matcheront, sauf si d'autres URLs sont matchées par un firewall précédemment configuré.

La clé *provider* réfère au nom du provider utilisé par le firewall « main ».

La clé *form_login* signifie que l'utilisateur s'authentifiera via un formulaire.

La clé *login_path* est le nom de la route sur laquelle l'utilisateur accède au formulaire. Il faudra donc créer le formulaire et l'afficher à l'utilisateur.[5]

La clé *check_login* est le nom de la route sur laquelle seront envoyé les données du formulaire. Il faudra que cette route existe en méthode POST.

La clé default_target_path est l'URL de redirection suite à une authentification réussie.

La clé *logout* n'est pas configurée par défaut. Lorsqu'un utilisateur se déconnecte depuis n'importe quel firewall, sa session devient invalide.

[4] Configuration Référence Sécurity Component - https://symfony.com/doc/3.0/reference/configuration/security.html
[5] Comment construire un formulaire de login - https://symfony.com/doc/current/security/form_login_setup.html

2) Guard et la personnalisation de l'authentification

Guard fait partie du composant Security. Son implémentation permet de modifier le comportement lors de l'authentification tout en gardant une démarche définie.[6]

Fichier de configuration

```
# config/packages/security.yaml
security:
# ...
firewalls:
        main:
            pattern: ^/private
anonymous: ~
provider: main
guard:
            authenticators:
              - app.security.login_form_authenticator
logout:
            path: /user/logout
success_handler: app.security.logout_success_handler
failure_handler: app.security.logout_failure_handler
```

La clé *guard* remplace *form_login* et mentionne le service *app.security.login_form_authenticator* pour gérer l'authentication.

Deux autres clés sont *success_handler et failure_handler*. Ces clés rattachent un service qui viendra personnaliser le comportement en cas de succès ou d'échec (cas rare) de la déconnexion.

Les clés *success_handler et failure_handler* sont aussi disponibles pour personnaliser l'authentification mais Guard est capable de le faire.

Pour rendre accessible ces services dans security.yaml, il faut préalablement les configurer dans le fichier services.yaml :

```
# config/service.yaml
services:

    app.security.login_form_authenticator:
        class: App\App\Security\LoginFormAuthenticator

app.security.logout_success_handler:
        class: App\App\Security\LogoutSuccessHandler

app.security.logout_failure_handler:
        class: App\App\Security\LogoutFailureHandler
```

[6] Personnaliser le système d'authentification avec Guard - https://symfony.com/doc/current/security/guard_authentication.html

7. Autorisation et Roles

Une fois l'authentification passée, Symfony va vérifier que l'utilisateur possède le Role requis en appelant la méthode getRoles() du User (Cf UserInterface[7]).

Chaque role doit obligatoirement commencé par *« ROLE_ »*, sinon le rôle ne sera pas reconnu par le Framework. Autre detail, un utilisateur peut posséder plusieurs rôles, il est donc stocké dans un tableau.

Exemple d'implementation dans une entités :

```php
private $roles = [];

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

return array_unique($roles);
}
```

1) La hiérarchie des Roles

La hiérarchie des rôles permet de donner les droits de plusieurs rôles à un seul par héritage.

```yaml
# config/packages/security.yaml
security:
# ...
role_hierarchy:
    ROLE_ADMIN:       ROLE_USER
ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

L'utilisateur avec le ROLE_ADMIN aura aussi les droits du ROLE_USER. Et l'utilisateur avec ROLE_SUPER_ADMIN aura automatiquement les rôles ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH et ROLE_USER.

[7] Symfony API UserInterface - https://api.symfony.com/3.4/Symfony/Component/Security/Core/User/UserInterface.html

2) Le contrôle des accès

La manière la plus élémentaire pour sécuriser une partie de l'application est de sécuriser une URL entière.

```
# config/packages/security.yaml
security:
# ...
firewalls:
        main:
# ...
access_control:
        - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/, roles: ROLE_USER }
```

Dans cet exemple, toutes les URLs requiert le ROLE_USER.

Le role IS_AUTHENTICATED_ANONYMOUSLY signifie « tous les utilisateurs » jusqu'à ce qu'ils soient authentifié. Ici nous autorisons l'URL « /login » d'être accessible par tous les utilisateurs, sinon l'accès serait refusé par la deuxième contrainte qui restreints toutes les URLs pour les ROLE_USER. Il est important de placer l'URL « /login » avant l'URL « / », sinon l'URL « /login » sera toujours interceptée et restreindra l'accès. Les URLs les plus précises doivent donc toujours être placées en premier.

Les URLs qui commencent par ^ signifie que seul les URL qui commencent par « ^/admin », admin seront restreint.

Les URLs sans ^, restreindra les URLs qui contiennent la chaîne de caractère. Par exemple avec /admin, les URL /foo/admin et /admin/foo seront restreintes.

3) Les Voters

Les voters[8] interviennent pour répondre à des besoins plus spécifiques comme par exemple, donner l'accès à certains articles d'un blog ou non en fonction de leur date de publication, ou parce que l'utilisateur est l'éditeur de cet article. Ces conditions sont déterminées par le développeur dans le voter.

[8] Les Voters - https://symfony.com/doc/current/security/voters.html

8. Manipuler la sécurité au travers de l'application

Depuis un controller

```
public function index()
{
// ici vous aurez fait le neccessaire pour authentifier l'utilisateur
   // lance une AccessDeniedException si l'utilisateur d'as pas le rôle
$this->denyAccessUnlessGranted('ROLE_MAXI_USER');

// retourne l'objet User ou null s'il n'est pas authentifié
$user = $this->getUser();
}
```

Depuis un service

```
// src/Service/ExampleService.php
use Symfony\Component\Security\Core\Security;

class ExampleService
{
private $security;

public function __construct(Security $security)
   {
// Evitez d'appeler getUser() dans le constructeur,
      // l'authentification peut ne pas avoir été terminée.
$this->security = $security;
}

public function someMethod()
   {
// retourne l'objet User ou null s'il n'est pas authentifié
$user = $this->security->getUser();
$user;
}
}
```

Depuis un template

```
{% if is_granted('ROLE_USER') %}
<p>Email: {{ app.user.email }}</p>
{% endif %}
```

Dans Twig, l'utilisateur est accessible via app.user