

**Niveau :** STS SIO 2<sup>ème</sup> année☐ Cours ☐ TD ☒ TP

1/30

**Module :** SLAM5 – Solutions applicatives**Intitulé :** Compléments sur Symfony 2**Durée :** 4h00

**Objectifs :**

- ✓ Sécuriser son site Symfony
- ✓ Mise en place de l'authentification
- ✓ Programmer au sein d'un Framework



## Contenu

<b>La sécurité sous Symfony2 .....</b>	<b>2</b>
Les notions d'authentification et d'autorisation .....	2
Processus général .....	3
Le fichier de configuration .....	4
<b>Mise en oeuvre .....</b>	<b>6</b>
Mettre en place un pare-feu .....	6
Définir une méthode d'authentification .....	7
Créer un formulaire de connexion .....	9
Définir les autorisations .....	13
Générer les entités User et Role .....	19
Mettre à jour le fichier de configuration de la sécurité .....	27
<b>Ressources .....</b>	<b>30</b>

# La sécurité sous Symfony2

2/30

## Les notions d'authentification et d'autorisation

- **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 n'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.

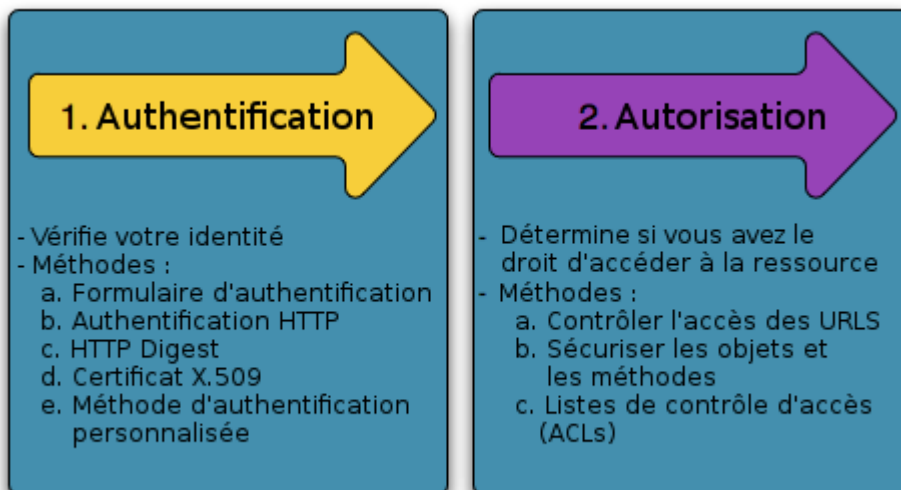
Ce qui gère l'authentification dans Symfony2 s'appelle un firewall.

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.

- **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 Symfony2 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 sujets. Seuls les membres disposant des droits d'administrateur le peuvent, ce que l'**Access control** va vérifier.



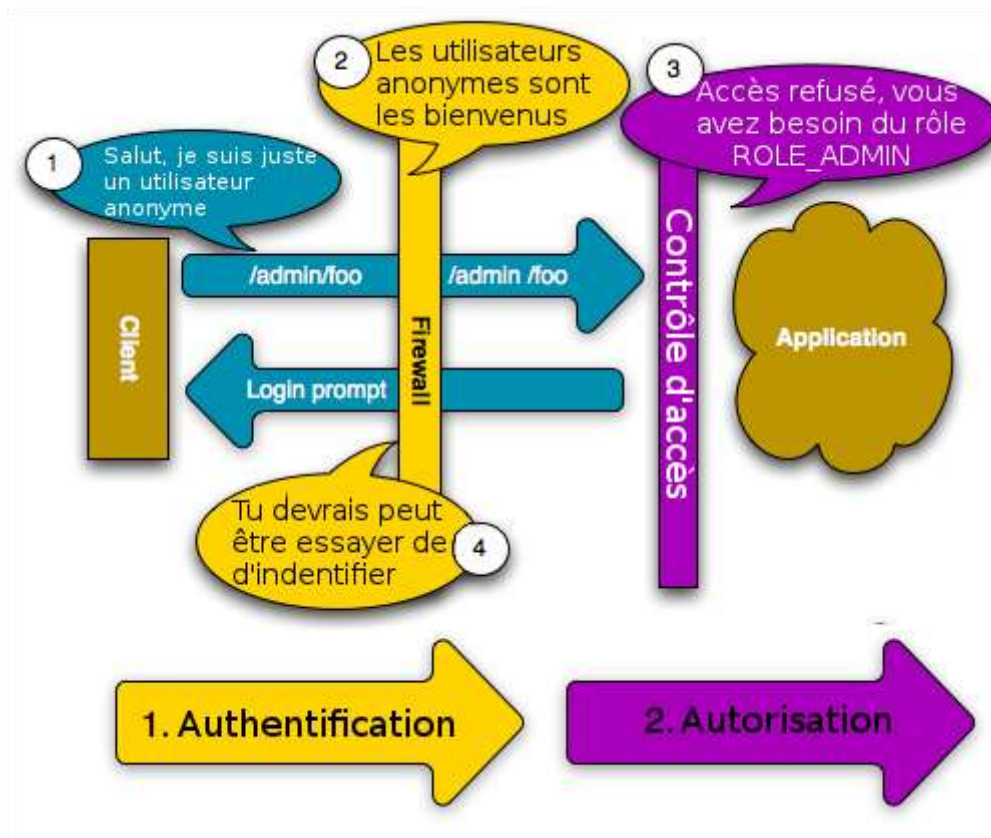
## Processus général

3/30

Lorsqu'un utilisateur tente d'accéder à une ressource protégée, le processus est finalement toujours le même, le voici :

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 envoie la requête initiale;
6. Le contrôle d'accès vérifie les droits de l'utilisateur, et autorise ou non l'accès à la ressource protégée.

Illustration : tentative de connexion à une partie sécurisée par un utilisateur anonyme



## Le fichier de configuration

Il s'agit du fichier `security.yml` se trouvant dans le répertoire `/app/config`

- Ouvrez ce fichier puis supprimer les sections `demo_login` et `demo_secured_area` qui se trouve sous `firewalls`.

Votre fichier doit maintenant ressembler à ceci :

```
security:
  encoders:
    Symfony\Component\Security\Core\User\User: plaintext

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

  providers:
    in_memory:
      memory:
        users:
          user: { password: userpass, roles: [ 'ROLE_USER' ] }
          admin: { password: adminpass, roles: [ 'ROLE_ADMIN' ] }

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

  access_control:
    #- { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY, requires_channel: https }
```

### Section encoders

Un encodeur est un objet qui encode les mots de passe de vos utilisateurs. Cette section de configuration permet donc de modifier l'encodeur utilisé pour vos utilisateurs, et donc la façon dont sont encodés les mots de passe dans votre application.

Ici, `plaintext` n'encode rien et laisse les mots de passe en clair. Pour l'instant on laisse `plaintext`, mais plus tard on utilisera un vrai encodeur, de type `sha512`.

### Section role\_hierarchy

La notion de « rôle » est au centre du processus d'autorisation. On assigne un ou plusieurs rôles à chaque utilisateur, et chaque ressource nécessite un ou plusieurs rôles. Ainsi, lorsqu'un utilisateur tente d'accéder à une ressource, le contrôleur d'accès vérifie s'il dispose du ou des rôles requis par la ressource. Si c'est le cas, l'accès est accordé. Sinon, l'accès est refusé.

Cette section définit la hiérarchie des rôles. Ainsi, le rôle `ROLE_USER` est compris dans le rôle `ROLE_ADMIN`. Cela signifie que si votre page requiert le rôle `ROLE_USER`, et qu'un utilisateur disposant du rôle `ROLE_ADMIN` tente d'y accéder, il sera autorisé, car en disposant du rôle d'administrateur, il dispose également du rôle `ROLE_USER`.

Les noms des rôles n'ont pas d'importance, si ce n'est qu'ils doivent commencer par « `ROLE_` ».

## Section providers

Un provider est un fournisseur d'utilisateurs. Les firewalls s'adressent aux providers pour récupérer les utilisateurs pour les identifier.

Actuellement, un seul fournisseur est défini : `in_memory`. Ce fournisseur est particulier puisque dans ce cas les utilisateurs sont listés directement dans ce fichier de configuration (il s'agit des utilisateurs user et admin).

C'est un fournisseur de développement utilisé pour tester la sécurité sans avoir à créer une table de base de données.

## Section firewalls

Un firewall (ou pare-feu) cherche à vérifier que vous êtes bien celui que vous prétendez être. Ici, seul le pare-feu `dev` est défini (nous avons supprimé les autres pare-feu de démonstration). Ce pare-feu permet de désactiver la sécurité sur certaines URL.

## Section access\_control

L'Access control va s'occuper de déterminer si le visiteur a les bons droits (rôles) pour accéder à la ressource demandée. Il y a différents moyens d'utiliser les contrôles d'accès :

- Soit dans ce fichier de configuration, en appliquant des règles sur des URL. On sécurise ainsi un ensemble d'URL en une seule ligne, par exemple toutes celles qui commencent par `/admin`.
- Soit directement dans les contrôleurs, en appliquant des règles sur les méthodes des contrôleurs. On peut ainsi appliquer des règles différentes selon des paramètres.

## Mise en oeuvre

6/30

### Mettre en place un pare-feu

- Dans le fichier `/app/config/security.yml`, créez un pare-feu nommé « main » comme ci-après :

```
security:
  encoders:
    Symfony\Component\Security\Core\User\User: plaintext

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

  providers:
    in_memory:
      memory:
        users:
          user: { password: userpass, roles: [ 'ROLE_USER' ] }
          admin: { password: adminpass, roles: [ 'ROLE_ADMIN' ] }

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

    main:
      pattern: ^/
      anonymous: true

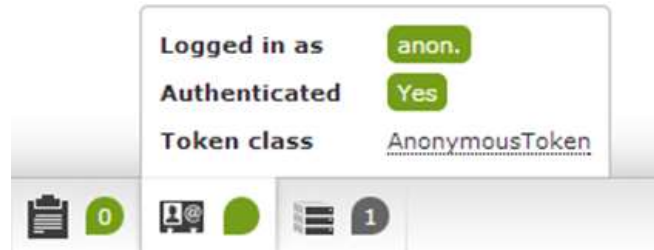
  access_control:
    #- { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY, requires_char
```

- **main** est le nom du pare-feu. Il s'agit juste d'un identifiant unique, vous pouvez mettre ce que vous voulez.
- **pattern: ^/** est un masque d'URL. Cela signifie que toutes les URL commençant par « / » (c'est-à-dire notre site tout entier) sont protégées par ce pare-feu. On dit qu'elles sont derrière le pare-feu **main**.
- **anonymous: true** accepte les utilisateurs anonymes. Nous protégerons nos ressources grâce aux rôles.

Il y a une hiérarchie dans les pare-feu : si une règle est définie dans un pare-feu, et que la règle inverse est définie dans un pare-feu se situant plus bas dans le fichier, c'est la règle du premier pare-feu qui sera appliquée.

- Connectez-vous à votre site :

Vous pouvez voir dans la barre d'outils en bas que vous êtes authentifiés en tant qu'anonyme, comme sur la figure suivante.



Si nous avons mis `anonymous` à `false`, vous n'auriez pas pu accéder à votre page.

## Définir une méthode d'authentification

La méthode d'authentification est obligatoire pour que le pare-feu puisse fonctionner. En effet le pare-feu a besoin de savoir comment vérifier si les utilisateurs sont identifiés et doit donc savoir où trouver les utilisateurs.

Nous allons utiliser un formulaire HTML standard pour l'authentification.

Au niveau du pare-feu, il faut donc ajouter l'option `form_login`.

➤ Modifiez votre fichier de configuration comme ci-après :

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

  main:
    pattern: ^/
    anonymous: true
    provider: in_memory
    form_login:
      login_path: login
      check_path: login_check
    logout:
      path: logout
      target: /login
```

### Explications

- **provider: in\_memory** est le fournisseur d'utilisateurs pour ce pare-feu. Comme je vous l'ai mentionné précédemment, un pare-feu a besoin de savoir où trouver ses utilisateurs, cela se fait par le biais de ce paramètre. La valeur `in_memory` correspond au fournisseur défini dans la section `providers` qu'on a vu précédemment.
- **form\_login** est la méthode d'authentification utilisée pour ce pare-feu. Elle correspond à la méthode classique, via un formulaire HTML. Ses options sont les suivantes :
  - **login\_path: login** correspond à la route du formulaire de connexion. En effet, ce formulaire est bien disponible à une certaine adresse, il s'agit ici de la route `login`, que nous définissons juste après.

- **check\_path: login\_check** correspond à la route de validation du formulaire de connexion, c'est sur cette route que seront vérifiés les identifiants renseignés par l'utilisateur sur le formulaire précédent.
- **logout** rend possible la déconnexion. En effet, par défaut il est impossible de se déconnecter une fois authentifié. Ses options sont les suivantes :
  - **path** est le nom de la route à laquelle le visiteur doit aller pour être déconnecté. On va la définir plus loin.
  - **target** est l'URL vers laquelle sera redirigé le visiteur après sa déconnexion.

Lorsque le pare-feu initie le processus d'authentification, il va rediriger l'utilisateur sur le formulaire de connexion (la route `login`). Le formulaire devra envoyer les valeurs vers la route (ici, `login_check`) qui va prendre en charge la soumission du formulaire.

Nous nous occupons du formulaire, mais c'est le système de sécurité de Symfony2 qui va s'occuper de la soumission de ce formulaire. Concrètement, nous allons définir un contrôleur à exécuter pour la route `login`, mais pas pour la route `login_check` ! Symfony2 va attraper la requête du visiteur sur la route `login_check`, et gérer lui-même l'authentification. En cas de succès, le visiteur sera authentifié. En cas d'échec, Symfony2 le renvoie vers le formulaire de connexion.

➤ Ajoutez les routes suivantes au routeur principal (`/app/config/routing.yml`) :

```
# ...
login:
    pattern:  /login
    defaults: { _controller: "SioUserBundle:Security:login" }

login_check:
    pattern:  /login_check

logout:
    pattern:  /logout
```

Comme vous pouvez le voir, on ne définit pas de contrôleur pour les routes `login_check` et `logout`. Symfony2 va attraper tout seul les requêtes sur ces routes.



## Créer un formulaire de connexion

### ➤ Générez le bundle SioUserBundle

Vous obtenez une erreur à la fin sur le routage : ceci est normal puisque nous avons déjà créé une route pour ce bundle.

Avant de poursuivre, vous allez faire un peu de ménage dans ce bundle :

- Supprimer le dossier Controller situé dans le dossier SioUserBundle/Test/
- Supprimer le dossier Default situé dans SioUserBundle/Resources/views/
- Supprimer le fichier routing.yml situé dans SioUserBundle/Resources/config/
- Renommez DefaultController.php situé dans SioUserBundle/Controller/ sous le nom SioUserBundle/Controller/SecurityController.php

### ➤ Gérer le contrôleur

Ouvrez le contrôleur SecurityController.php et modifiez-le comme ci-dessous; n'oubliez pas de modifier le nom de la classe :

```
<?php

namespace Sio\UserBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\Security\Core\SecurityContext;

class SecurityController extends Controller
{
    public function loginAction()
    {
        // Si le visiteur est déjà identifié, on le redirige vers l'accueil
        if ($this->get('security.context')->isGranted('IS_AUTHENTICATED_REMEMBERED')) {
            return $this->redirect($this->generateUrl('Sio_accueil'));
        }

        $request = $this->getRequest();
        $session = $request->getSession();

        // On vérifie s'il y a des erreurs d'une précédente soumission du formulaire
        if ($request->attributes->has(SecurityContext::AUTHENTICATION_ERROR)) {
            $error = $request->attributes->get(SecurityContext::AUTHENTICATION_ERROR);
        } else {
            $error = $session->get(SecurityContext::AUTHENTICATION_ERROR);
            $session->remove(SecurityContext::AUTHENTICATION_ERROR);
        }

        return $this->render('SioUserBundle:Security:login.html.twig', array(
            // Valeur du précédent nom d'utilisateur entré par l'internaute
            'last_username' => $session->get(SecurityContext::LAST_USERNAME),
            'error'          => $error,
        ));
    }
}
```

## ➤ Créer la vue

Créez un dossier `Security` dans `/src/Sio/UserBundle/Resources/views`  
 Puis créez le fichier de vue `login.html.twig` suivant dans ce nouveau répertoire :

```
{% extends "::base.html.twig" %}

{% block title %}Connexion{% endblock %}
{% block pagetitle %}Accès{% endblock %}

{% block body %}
    <br/><br/>
    <div class="row">
        <div class="col-md-2"></div>
        <div class="col-md-8">
            {# S'il y a une erreur, on l'affiche #}
            {% if error %}
                <div class="alert alert-danger">{{ error.message }}</div>
            {% endif %}
        </div>
    </div>
    <div class="row">
        <div class="col-md-4"></div>
        <div class="col-md-4">
            {# Le formulaire, avec URL de soumission vers la route login_check #}
            <form action="{{ path('login_check') }}" method="post">

                <div class="form-group">
                    <label for="username">Login :</label>
                    <input type="text" class="form-control" id="username"
                        name="_username" value="{{ last_username }}" />
                </div>

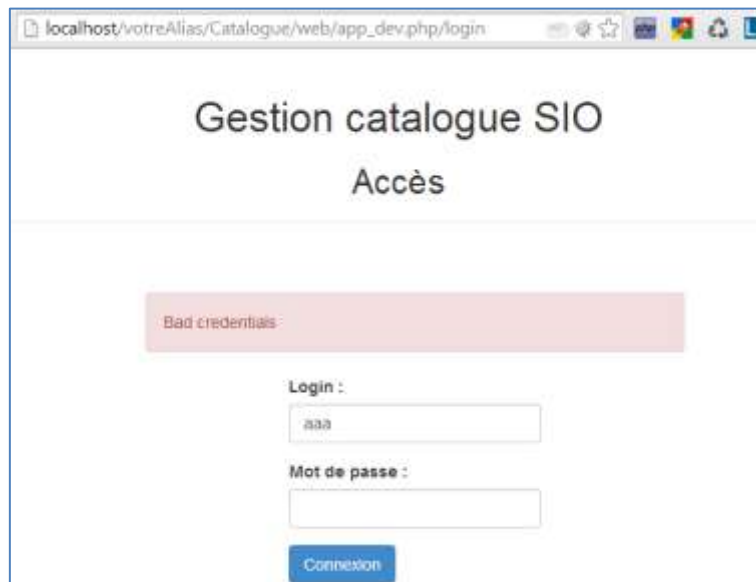
                <div class="form-group">
                    <label for="password">Mot de passe :</label>
                    <input type="password" class="form-control"
                        id="password" name="_password" />
                </div>

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

On teste :



Et si vous entrez de mauvais identifiants :



Si vous entrez les bons identifiants (user et userpass), vous êtes bien redirigés vers la page d'accueil.

Et vous pouvez vérifier que vous êtes effectivement authentifié :



Pour vous déconnecter, il suffit de taper l'url :

[http://localhost/votreAlias/Catalogue/web/app\\_dev.php/logout](http://localhost/votreAlias/Catalogue/web/app_dev.php/logout)

## ➤ Récupérer l'utilisateur courant

Pour récupérer les informations sur l'utilisateur courant, qu'il soit anonyme ou non, il faut utiliser le service `security.context`.

Voici comment l'utiliser **depuis un contrôleur** :

```
<?php
[...]
$user = $this->getUser();

if ($user === null) {
    // L'utilisateur est anonyme ou l'URL n'est pas derrière un pare-feu
} else {
    // $user est une instance de la classe User
[...]
```

Voici comment l'utiliser **depuis une vue Twig** :

```
{% if app.user is not null %}
    {{ app.user.username }} est connecté
{% endif %}
```

**Exemple** : nous allons afficher le nom de l'utilisateur connecté sur toutes les pages.

Modifiez le template de base : `/app/Resources/views/base.html.twig`

```
<!DOCTYPE html>
<html>
    <head>

    <body>

        <div class="container">
            <div class="nav navbar-nav navbar-right">
                {% if app.user is not null %}
                    {{ app.user.username }} est connecté
                {% endif %}
            </div>
        </div>

        <div class="container">
            <div class="page-header">
                <h1 class="text-center strong">Gestion catalogue SIO</h1>
                <h2 class="text-center">{% block pagetitle %}{% endblock %}</h2>
            </div>
            <div class="container">
                {% block body %}{% endblock %}
            </div>
        </div>

        {% block javascripts %}{% endblock %}

    </body>
</html>
```

Voilà le résultat :



## Définir les autorisations

Une fois l'utilisateur authentifié, l'autorisation commence. L'autorisation fournit un moyen de décider si un utilisateur peut accéder à une ressource (une URL, un objet du modèle, un appel de méthode...).

Cela fonctionne en **assignant des rôles à chaque utilisateur**, et d'ensuite en requérant différents rôles pour différentes ressources.

Le processus d'autorisation comporte 2 aspects :

1. Un utilisateur possède un ensemble de rôles
2. Une ressource requiert un rôle spécifique pour être atteinte

Il y a deux moyens pour sécuriser une application :

- En utilisant le service **security.context**  
Cette méthode permet de définir des privilèges dans un contrôleur (sur une action d'un contrôleur) ou dans une vue
- En utilisant les **contrôles d'accès** (access control)  
Avec cette méthode, il est possible de sécuriser des URL's, des adresses IP, un canal (ex : https), ...

Nous n'allons pas étudier toutes les méthodes. Nous nous contenterons des suivantes :

- La sécurisation au niveau des URL grâce à Access control
- La sécurisation au niveau des actions d'un contrôleur et d'une vue grâce au service security.context

Pour plus d'informations, rendez-vous sur le site officiel, à la page suivante :

[http://symfony.com/fr/doc/current/cookbook/security/securing\\_services.html](http://symfony.com/fr/doc/current/cookbook/security/securing_services.html)

## ❖ Définir les rôles et les utilisateurs

Afin que vous ayez une bonne vision de la gestion des rôles et des utilisateurs, nous allons créer 3 rôles et un utilisateur pour chacun de ces rôles.

- L'employé aura le droit d'accéder à l'application mais ne pourra pas modifier les données (pas d'ajout ni de modification ni de suppression).
- Le gestionnaire aura les mêmes droits que l'employé avec en plus la possibilité d'ajouter et de modifier.
- L'administrateur aura tous les droits (donc la suppression en plus).

Il faut tout d'abord définir la **hiérarchie de rôles**.

Ensuite on définit les **utilisateurs** et leur affecte un rôle.

Modifiez votre fichier `/app/config/security.yml` ainsi :

```
security:
    encoders:
        Symfony\Component\Security\Core\User\User: plaintext

    role_hierarchy:
        # Un gestionnaire hérite des droits d'employé
        ROLE_GESTIONNAIRE: [ROLE_EMPLOYE]
        # Un admin hérite des droits d'employé et de gestionnaire
        ROLE_ADMIN: [ROLE_EMPLOYE, ROLE_GESTIONNAIRE]
        # On conserve les droits de super admin en l'état
        ROLE_SUPER_ADMIN: [ROLE_EMPLOYE, ROLE_GESTIONNAIRE, ROLE_ADMIN,
        ROLE_ALLOWED_TO_SWITCH]

    providers:
        in_memory:
            memory:
                users:
                    empl: { password: empl, roles: [ 'ROLE_EMPLOYE' ] }
                    gest: { password: gest, roles: [ 'ROLE_GESTIONNAIRE' ] }
                    admin: { password: admin, roles: [ 'ROLE_ADMIN' ] }
                    root: { password: root, roles: [ 'ROLE_SUPER_ADMIN' ] }

    firewalls:
[...]
```

## ❖ Sécuriser l'application

### ○ Utiliser les contrôles d'accès

Nous allons voir ici la méthode qui consiste à sécuriser le site au niveau d'une URL.

La façon la plus simple pour sécuriser une partie de votre application est de sécuriser un masque d'URL au complet.

Tout se passe dans la section `access_control` du fichier `/app/config/security.yml`.

On utilise des **expressions régulières** pour identifier les URL's à sécuriser.

```
security:
[...]
  firewalls:
[...]
    access_control:
      - { path: /delete, roles: ROLE_ADMIN }
      - { path: /new, roles: ROLE_GESTIONNAIRE }
      - { path: /create, roles: ROLE_GESTIONNAIRE }
      - { path: /edit, roles: ROLE_GESTIONNAIRE }
      - { path: /update, roles: ROLE_GESTIONNAIRE }
      - { path: ^/produit, roles: ROLE_EMPLOYE }
      - { path: ^/categorie, roles: ROLE_EMPLOYE }
```

- { path: /delete, roles: ROLE\_ADMIN }

Cette ligne indique que pour toutes les URL's **contenant** /delete, il faut avoir au minimum le rôle d'administrateur.

Ainsi, l'url `http://.../web/app_dev.php/produit/1/delete` sera accessible à un administrateur uniquement.

- { path: ^/produit, roles: ROLE\_EMPLOYE }

Cette ligne indique que pour toutes les URL's **commençant** par /produit, il faut avoir au minimum le rôle d'administrateur.

Ainsi, l'url `http://.../web/app_dev.php/produit/new` sera accessible à un employe, à un gestionnaire et à l'administrateur. Par contre, un anonyme ne peut pas accéder à cette page.

Idem pour `http://.../web/app_dev.php/produit/1/edit`, ...

Par contre, rien n'est spécifié pour une URL qui serait par exemple :

Idem pour `http://.../web/app_dev.php/produits/produit/1`, ...

Il faut que le chemin commence par /produit. C'est le « ^ » qui précise ici que l'URL doit commencer par la chaîne spécifiée.

Testez votre application en vous connectant sous différentes identités.

On pourrait affiner, les expressions régulières sont très puissantes. Mais ce n'est pas l'objet du TP.



Soyez juste prudent lorsque vous utilisez la sécurité au niveau des URL's. Il ne faut jamais sécuriser la page de login (path ^/login) ni la page vers laquelle on est renvoyé en cas d'erreur de connexion.

- **Utiliser le service `security.context` dans un contrôleur**

16/30

Il est possible également de définir les autorisations à l'intérieur des contrôleurs.

Nous allons mettre en place les mêmes contrôles que précédemment, mais en le faisant directement au niveau des actions des contrôleurs.

Avant de commencer, vous allez mettre en commentaires toutes les lignes de la section `access_control` du fichier `/app/config/security.yml` en plaçant un `#` en début de ligne :

```
security:
[...]
  providers:
[...]
  firewalls:
[...]
  access_control:
    # - { path: /delete, roles: ROLE_ADMIN }
    # - { path: /new roles: ROLE_GESTIONNAIRE }
    # - { path: /create, roles: ROLE_GESTIONNAIRE }
[...]
```

Enregistrez.

Déconnectez-vous de l'application si besoin. Puis vérifiez que vous avez accès à n'importe quelle page en tant qu'anonyme.

Nous allons nous intéresser uniquement à la gestion des produits.

Depuis votre contrôleur **ProduitController**, il vous faut accéder au service `security.context` et appeler la méthode `isGranted`, tout simplement.

```
<?php

namespace Sio\GestionCatalogueBundle\Controller;
[...]
```

```
use Symfony\Component\HttpKernel\Exception\AccessDeniedHttpException;
```

```
class ProduitController extends Controller
{
    public function indexAction()
    {
        // Si l'utilisateur ne dispose pas du rôle ROLE_EMPLOYE
        if ($this->get('security.context')->isGranted('ROLE_EMPLOYE') === false)
        {
            throw new AccessDeniedHttpException('Accès limité - Il faut être authentifié');
        }

        // La suite est inchangée
        [...]
    }
}
```

```
.../...
```



```
.../...  
  
public function createAction(Request $request)  
{  
    // Si l'utilisateur ne dispose pas du rôle ROLE_GESTIONNAIRE  
    if (!$this->get('security.context')->isGranted('ROLE_GESTIONNAIRE') === false)  
    {  
        throw new AccessDeniedHttpException('Accès limité - Il faut  
            être authentifié');  
    }  
  
    // La suite est inchangée  
    [...]  
}  
  
[...]  
}
```

A vous !

Continuez pour les méthodes : `newAction`, `showAction`, `editAction`, `updateAction`, `deleteAction`.

Les méthodes suivantes ne sont pas concernées puisqu'elles ne sont pas référencées dans le fichier de routes : `createcreateForm`, `createDeleteForm` et `createEditForm`.

Testez.

Vous voyez, c'est facile à mettre en œuvre. Cela permet d'être plus précis et évite de se prendre la tête avec les expressions régulières.

### ○ Utiliser le service `security.context` dans une vue

Cette méthode est pratique pour afficher du contenu différent selon les rôles des utilisateurs.

Par exemple, la vue `index.html.twig` (liste des produits) est accessible par tous les utilisateurs authentifiés (non anonymes).

Par contre, un employé n'a pas accès à la création ni la modification d'un produit.

Plutôt que de lui afficher des liens qui le renverront vers une page « 404 – Not found », il est préférable ne pas lui afficher les liens.

Twig dispose d'une fonction `is_granted()` qui est en réalité un raccourci pour exécuter la méthode `isGranted()` du service `security.context`

Voici comment faire :

Fichier `/src/Sio/CatalogueBundle/Resources/views/Produit/index.html.twig`

```
{% extends '::base.html.twig' %}

{% block title %}
Produit
{% endblock %}

{% block pagetitle %}
Liste des produits
{% endblock %}

{% block body -%}


```

Vérifier que vous n'avez plus le lien « Créer un nouveau produit » lorsque vous êtes connecté en tant que « empl » mais qu'il apparaît bien si vous êtes connecté en tant que « gest » ou « admin ».

Modifiez les autres vues afin de masquer les liens non autorisés. Testez avec les 3 comptes utilisateurs.

## Gérer les utilisateurs dans la base de données

19/30

Jusque-là nous avons géré les utilisateurs dans le fichier de configuration de la sécurité. Mais ce n'est ni très pratique ni vraiment sécurisé.

Il est donc préférable de les gérer dans une base de données. Nous allons voir comment faire.

### Générer les entités User et Role

#### 1. Création de l'entité Role

Générez une entité **Role** dans le bundle **SioUserBundle** avec une seule propriété **role** de type string longueur 50.

Attention, le nom de cette propriété doit être respecté car il est utilisé par la couche sécurité de Symfony.

#### 2. Création de l'entité User

Générez une entité **User** dans le bundle **SioUserBundle** avec les propriétés suivantes :

- **username** : c'est l'identifiant de l'utilisateur au sein de la couche sécurité. Cela n'empêchera pas d'utiliser un id auto incrémenté pour l'entité  
→ string 50
- **password** : le mot de passe  
→ string 255  
*Nous allons crypter les mots de passe en sha512, il faut donc de « la place » !*
- **salt** : le sel, pour encoder le mot de passe  
→ string 50

Attention, les noms de ces propriétés doivent être respectés car ils sont utilisés par la couche sécurité de Symfony.

Avant de générer les tables dans la base de données, il faut apporter quelques modifications à nos entités.

#### 3. Modification de l'entité Role

Pour que notre entité Role soit reconnue par la couche sécurité de Symfony, elle doit obligatoirement implémenter l'interface **RoleInterface** et par conséquent définir les méthodes suivantes de cette interface :

Interface  
Symfony\Component\Security\Core\Role\RoleInterface

interface **RoleInterface**

RoleInterface represents a role granted to a user.  
A role must either have a string representation or it needs to be explicitly supported.

**Methods**

string|null **getRole()**  
Returns the role.

Comme nous avons déclaré une propriété « role », la méthode `getRole()` (le getter) a bien été générée.

Il ne reste donc qu'à indiquer que la classe implémente l'interface `RoleInterface`.

Ouvrez le fichier `src/Sio/UserBundle/Entity/Role.php` généré et apportez les modifications suivantes :

```
<?php

namespace Sio\UserBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\Role\RoleInterface;

/**
 * Role
 *
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="Sio\UserBundle\Entity\RoleRepository")
 */
class Role implements RoleInterface
{
```

#### 4. Modification de l'entité User

Pour que notre entité `User` soit reconnue par la couche sécurité de Symfony, elle doit obligatoirement implémenter l'interface **`UserInterface`** et par conséquent définir les méthodes suivantes de cette interface :

Interface <b>Symfony\Component\Security\Core\User\UserInterface</b>
<b>interface UserInterface</b>  Represents the interface that all user classes must implement. This interface is useful because the authentication layer can deal with the object through Regardless of how your user are loaded or where they come from (a database, configuration, ...) UserProviderInterface
<b>Methods</b>
Role[] <b>getRoles()</b> Returns the roles granted to the user.
string <b>getPassword()</b> Returns the password used to authenticate the user.
string <b>getSalt()</b> Returns the salt that was originally used to encode the password.
string <b>getUsername()</b> Returns the username used to authenticate the user.
void <b>eraseCredentials()</b> Removes sensitive data from the user.
Boolean <b>equals(UserInterface \$user)</b> Returns whether or not the given user is equivalent to <i>this</i> user.

Compte-tenu des propriétés que nous avons créées dans notre entité, 3 des méthodes de l'interface ont déjà été générées (ce sont les getters de notre classe).

Il reste à implémenter les méthodes `getRoles()`, `eraseCredentials()` et `equals()`.

Par ailleurs, nous devons pouvoir associer nos utilisateurs à des rôles. Il faut donc définir une relation entre nos entités : il s'agit ici d'une relation `ManyToMany` (1 utilisateur peut avoir plusieurs rôles, un rôle peut être affecté à plusieurs utilisateurs). Au niveau des entités, on ne gère la relation que dans un seul sens : le `User` connaît ses rôles. Nous avons donc besoin d'une propriété `$roles` de type collection : le type Doctrine à utiliser est `ArrayCollection`.

Enfin, il faut implémenter l'interface `serializable` afin de permettre à la classe `User` d'être sérialisable. En effet, en fonction de la configuration de PHP, le fait de ne pas créer les méthodes permettant de sérialiser/désérialiser l'objet `User` provoque un plantage de l'application.

Ouvrez le fichier `src/Sio/UserBundle/Entity/User.php` généré et apportez les modifications suivantes :

- Ajoutez une référence aux namespaces utilisés

```
<?php

namespace Sio\UserBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\User\EquatableInterface;
use Doctrine\Common\Collections\ArrayCollection;
```

- Indiquez que notre classe User implémente les interfaces UserInterface et Serializable

```
/**
 * User
 *
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="Sio\UserBundle\Entity\UserRepository")
 */
class User implements UserInterface, \Serializable
{
```

Rappel : le « \ » placé devant Serializable indique qu'il s'agit d'une interface du langage PHP et non de Symfony ou l'un de ses composants.

- Définir l'attribut username comme étant unique, car c'est l'identifiant qu'utilise la couche sécurité

```
/**
 * @var string
 *
 * @ORM\Column(name="username", type="string", length=50, unique=true)
 */
private $username;
```

- Déclarer la propriété \$roles qui va permettre de gérer la relation entre un utilisateur et ses rôles

```
/**
 * @var ArrayCollection
 *
 * @ORM\ManyToMany(targetEntity="Role")
 */
private $roles;
```

- Ajoutez un constructeur permettant d'instancier le tableau de rôles :

```
public function __construct()
{
    $this->roles = new ArrayCollection();
}
```

- Définir la méthode `getRoles()` de l'interface `ManagerInterface`

```
/**
 * @inheritDoc ManagerInterface
 */
public function getRoles()
{
    return $this->roles->toArray();
}
```

Attention, ici il est nécessaire de transformer notre `arrayCollection` en `array` php.

- Définir les méthodes `eraseCredentials()` et `equals()` de l'interface `ManagerInterface`

```
/**
 * @inheritDoc ManagerInterface
 */
public function eraseCredentials()
{
}

/**
 * @inheritDoc ManagerInterface
 */
public function equals(MManagerInterface $user)
{
    return $this->username === $user->getUsername();
}
```

On crée une méthode `eraseCredentials()` vide car on ne s'en servira pas.

- Définir les méthodes `serialize()` et `unserialize()` de l'interface `Serializable`

```
/**
 * @inheritDoc \Serializable::serialize()
 */
public function serialize()
{
    return serialize(array(
        $this->id,
    ));
}

/**
 * @inheritDoc \Serializable::unserialize()
 */
public function unserialize($serialized)
{
    list (
        $this->id,
    ) = unserialize($serialized);
}
```

## 5. Mettre à jour l'entité User

Lancez la commande :

```
app/console doctrine:generate:entities Sio/UserBundle/Entity/User
```

Allez voir votre classe `User` : Doctrine a ajouté les méthodes `addRole()` et `removeRole()` à votre classe (car la propriété `$roles` a été déclarée avec le type `ArrayCollection`).

## 6. Générer les tables dans la base de données

Lancez la commande : `app/console doctrine:schema:update --force`

Et voilà, vos tables sont prêtes :

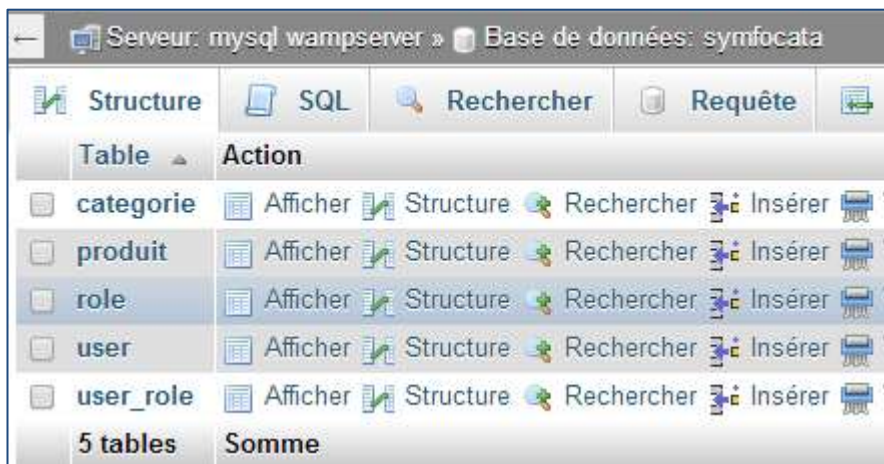


Table	Action
categorie	Afficher Structure Rechercher Insérer
produit	Afficher Structure Rechercher Insérer
role	Afficher Structure Rechercher Insérer
user	Afficher Structure Rechercher Insérer
user_role	Afficher Structure Rechercher Insérer
5 tables	Somme

Vous pouvez constater la présence d'une table `user_role` qui est la table associative représentant la relation entre vos tables `user` et `role`.

## 7. Générer des données

Nous allons utiliser le bundle `DoctrineFixtureBundle` puisqu'il est installé.

Voici le fichier de fixtures, que je vous donne pour une fois (récupérez le fichier `LoadUsersGroups.php`) et placez-le dans le bundle `SioUserBundle` après avoir créé les répertoires qui vont bien !



```

<?php
namespace Sio\UserBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Sio\UserBundle\Entity\User;
use Sio\UserBundle\Entity\Role;

use Symfony\Component\Security\Core\Encoder\MessageDigestPasswordEncoder;

class LoadUsersGroups implements FixtureInterface
{
    // Dans l'argument de la méthode load, l'objet $manager est
    // l'EntityManager
    public function load(ObjectManager $manager)
    {
        /**
         * Création des groupes
         */
        $role_employe = new Role();
        $role_employe->setRole("ROLE_EMPLOYE");
        $manager->persist($role_employe);
        $manager->flush();

        $role_gestionnaire = new Role();
        $role_gestionnaire->setRole("ROLE_GESTIONNAIRE");
        $manager->persist($role_gestionnaire);
        $manager->flush();

        $role_admin = new Role();
        $role_admin->setRole("ROLE_ADMIN");
        $manager->persist($role_admin);
        $manager->flush();

        $role_superadmin = new Role();
        $role_superadmin->setRole("ROLE_SUPER_ADMIN");
        $manager->persist($role_superadmin);
        $manager->flush();

        /**
         * Création des utilisateurs
         */
        $usr = new User();
        $usr->setUsername("balou");
        // On crée un salt pour améliorer la sécurité
        $usr->setSalt(md5(time()));
        // On crée un mot de passe (attention, il faut utiliser les mêmes paramètres
        // que dans le fichier security.yml, à savoir SHA512 avec 10 itérations.
        $encoder = new MessageDigestPasswordEncoder('sha512', true, 10);
        // On crée le mot de passe (nom de l'utilisateur) à partir de l'encodage choisi
        $password = $encoder->encodePassword('balou', $usr->getSalt());
        // On affecte le mot de passe à l'utilisateur
        $usr->setPassword($password);
        // On affecte le rôle à l'utilisateur
        $usr->addRole($role_employe);
        // On persiste l'utilisateur
        $manager->persist($usr);
        // On déclenche l'enregistrement
        $manager->flush();
    }
}
.../...

```

```

    $usr = new User();
    $usr->setUsername("dédé");
    $usr->setSalt(md5(time()));
    $encoder = new MessageDigestPasswordEncoder('sha512', true, 10);
    $password = $encoder->encodePassword('dédé', $usr->getSalt());
    $usr->setPassword($password);
    $usr->addRole($role_gestionnaire);
    $manager->persist($usr);
    $manager->flush();

    $usr = new User();
    $usr->setUsername("jojo");
    $usr->setSalt(md5(time()));
    $encoder = new MessageDigestPasswordEncoder('sha512', true, 10);
    $password = $encoder->encodePassword('jojo', $usr->getSalt());
    $usr->setPassword($password);
    $usr->addRole($role_admin);
    $manager->persist($usr);
    $manager->flush();
}

```

Lancez les fixtures : `php app/console doctrine:fixtures:load`

4 rôles ont été créés (les mêmes que ceux que l'on avait utilisés précédemment).

3 utilisateurs ont été créés : *balou*, *dédé* et *jojo*, avec chacun un rôle différent.

Nous aurions pu leur attribuer plusieurs rôles, mais comme nous avons défini une hiérarchie de rôles dans le fichier de configuration, cela n'est pas nécessaire.

Pour encoder les mots de passe nous utilisons la classe `MessageDigestPasswordEncoder` du namespace

`Symfony\Component\Security\Core\Encoder\MessageDigestPasswordEncoder`

Voici la documentation du constructeur de cette classe :

```

public __construct(string $algorithm = 'sha512', Boolean $encodeHashAsBase64 = true, integer $iterations = 5000)

```

Constructor.

Parameters		
string	\$algorithm	The digest algorithm to use
Boolean	\$encodeHashAsBase64	Whether to base64 encode the password hash
integer	\$iterations	The number of iterations to use to stretch the password hash

Vous pouvez donc constater que les mots de passe (identiques aux noms) ont été cryptés avec l'algorithme de cryptage `sha512`, en utilisant l'encodage base64, avec 10 itérations.

Voilà. Il reste à mettre à jour le fichier de configuration.

## Mettre à jour le fichier de configuration de la sécurité

27/30

Ouvrir le fichier `app/config/security.yml`

### ❖ 1ère étape : définir l'encodeur pour notre classe User

```
security:
  encoders:
    Symfony\Component\Security\Core\User\User: plaintext
    Sio\UserBundle\Entity\User:
      algorithm: sha512
      encode-as-base64: true
      iterations: 10
```

### ❖ 2<sup>ème</sup> étape : définir le fournisseur d'utilisateurs

Le fournisseur (**provider**) permet au pare-feu d'identifier et de récupérer les utilisateurs.

Un fournisseur d'utilisateurs est une classe qui implémente l'interface `UserProviderInterface` contenant 3 méthodes

- `loadUserByUsername($username)`, qui charge un utilisateur à partir d'un nom d'utilisateur ;
- `refreshUser($user)`, qui rafraîchit un utilisateur avec les valeurs d'origine ;
- `supportsClass()`, qui détermine quelle classe d'utilisateurs gère le fournisseur.

Symfony2 dispose de trois types de fournisseurs :

- `memory`, qui utilise les utilisateurs définis dans la configuration (c'est celui qu'on a utilisé jusqu'à maintenant)
- `entity`, qui utilise une entité pour fournir les utilisateurs (c'est celui qu'on va utiliser)
- `id`, qui permet d'utiliser un service quelconque en tant que fournisseur, en précisant le nom du service.

Définissez le provider pour votre classe User :

```
providers:
  main:
    entity: {class Sio\UserBundle\Entity\User, property: username }
```

*Supprimez le fournisseur `in_memory` existant*

❖ 3<sup>ème</sup> étape : indiquez au pare-feu le fournisseur à utiliser

```
firewalls:
  main:
    pattern: ^/
    anonymous: true
    provider: main
    form_login:
      login_path: /login
      check_path: /login_check
    logout:
      path: /logout
      target: /
```

Et voilà : la sécurité est en place.

Testez votre application en utilisant les différents utilisateurs.

Attention, seules les pages concernant les produits sont sécurisées puisque nous avons mis les tests en place dans le seul contrôleur Produit du bundle SioGestionCatalogueBundle.

Bon allez, juste pour le fun, on va ajouter un bouton de connexion et un bouton de déconnexion en haut de nos pages.



Ouvrez le fichier `app/Resources/views/base.html.twig` et apportez les modifications ci-dessous :

29/30

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>{% block title %}Catalogue SIO{% endblock %}</title>
    {% block stylesheets %}
      <link rel="stylesheet" href="{{ asset('css/bootstrap.css') }}"
        type="text/css" />
    {% endblock %}
    <link rel="icon" type="image/x-icon" href="{{ asset('favicon.ico') }}" />
  </head>
  <body>
    <div class="container">
      <div class="nav navbar-nav navbar-right">
        {% if app.user is not null %}
          {{ app.user.username }} est connecté&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~
          <a href="{{ path('logout') }}">Déconnexion</a>
        {% else %}
          <a href="{{ path('login') }}">Connexion</a>
        {% endif %}
      </div>
    </div>
  </div>
```

Testez.

Pour terminer, il faudrait prévoir les pages permettant de gérer les utilisateurs. Mais bon, vous avez toutes les connaissances requises pour le faire seul.

Sachez cependant qu'il existe des bundles tout prêts permettant de gérer les utilisateurs et l'authentification

Voir sur <https://packagist.org>, en cherchant le terme « user » ou encore <http://knpsbundles.com/> qui référence les bundles Symfony

**FOSUserBundle**, est l'un de ces bundles spécialisés dans la gestion des utilisateurs. Il est très utilisé par la communauté Symfony2 et préconisé dans la documentation officielle de Symfony, car il répond parfaitement au besoin basique d'un site internet : l'authentification des membres.

Vous trouverez de nombreuses documentations sur internet expliquant comment le mettre en œuvre.

Voilà, nous allons nous arrêter là en ce qui concerne Symfony.

Nous sommes loin d'avoir fait le tour de la question. Il reste encore plein de choses à découvrir : les services, le gestionnaire d'évènements, la personnalisation des pages d'erreurs, Assetic, mise en production, ...

Mais ces 5 TP vous ont permis de découvrir la puissance d'un Framework PHP et de comprendre tous les intérêts que l'on peut y trouver.

## Ressources

30/30

Il existe de nombreux tutos en ligne. Je vous redonne ici quelques références, que j'ai sélectionnées pour leur qualité :

- Le site officiel de Symfony, évidemment : <http://symfony.com/> et en particulier « The Symfony book » (en français, je vous rassure)  
<http://symfony.com/fr/doc/current/book/index.html>
- Le site officiel de Doctrine : <http://www.doctrine-project.org/>
- La documentation des API de Doctrine :  
Couche ORM : <http://www.doctrine-project.org/api/orm/2.0/index.html>  
Couche DBAL : <http://www.doctrine-project.org/api/dbal/2.0/index.html>
- La documentation de l'API de Symfony, pratique et indispensable :  
<http://api.symfony.com/2.5/index.html>
- Le tuto « Créer sa première application web en PHP avec Symfony2 » sur le site de Développez.com  
<http://j-place.developpez.com/tutoriels/php/creer-premiere-application-web-avec-symfony2/#LXIII>
- Le tuto « Développez votre site web avec le framework Symfony2 » sur le site de Open Classrooms, très bon tuto, très complet, et dont je me suis largement inspirée  
<http://fr.openclassrooms.com/informatique/cours/developpez-votre-site-web-avec-le-framework-symfony2>
- Le tuto « Création de sa première application Symfony2 » sur le site de iGestis, tuto clair et concis  
<https://iabsis.com/FR/article/21/Tuto-Creation-de-sa-premiere-application-Symfony2/>
- Le tutoriel pour Symfony2 en français sur le site Jobeet FR  
<http://jobeet.thuau.fr/>
- Et, pour la recherche de bundles  
<https://packagist.org>  
<http://knpbundles.com/>