

Symfony Sécurité

AYMEN SELLAOUTI

Introduction

- Un site est généralement décomposé en deux parties :
 - Partie publique : accessible à tous le monde
 - Partie privée : accessible à des utilisateurs particuliers.
 - Au sein même de la partie privée, certaines ressources sont spécifiques à des rôles ou des utilisateurs particuliers.

Nous identifions donc deux niveaux de sécurité :

Authentification

Autorisation

Authentication

- Processus permettant d'authentifier un utilisateur.
- Deux réponses possibles
 - Non authentifié : Anonyme.
 - Authentifié : membre

Security Bundle

- Le Bundle qui gère la sécurité dans Symfony s'appelle SecurityBundle.
- Si vous ne l'avez pas dans votre application, installer le via la commande

composer require security

Fichier de configuration security.yml

```
security:
  # https://symfony.com/doc/current/security/authenticator_manager.html
  enable_authenticator_manager: true
  # https://symfony.com/doc/current/security.html#c-hashing-passwords
  password_hashers:
    Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
  # https://symfony.com/doc/current/security.html#where-do-users-come-from-user-providers
  providers:
    users_in_memory: { memory: null }
  firewalls:
    dev:
      pattern: ^/(_(profiler|wdt)|css|images|js)/
      security: false
    main:
      lazy: true
      provider: users_in_memory
      # switch_user: true
  # Easy way to control access for large sections of your site
  # Note: Only the *first* access control that matches will be used
  access_control:
    # - { path: ^/admin, roles: ROLE_ADMIN }
```

Le Firewall qui gère la configuration de l'authentification de vos utilisateurs

Cette partie assure que le débogueur de Symfony ne soit pas bloqué

La classe user

- L'ensemble du système de sécurité est basé sur la classe **User** qui représente l'utilisateur de votre application.
- Afin de créer la classe **User**, utiliser la commande :
symfony console make:user
- Si vous n'avez pas le MakerBundle, installer le.
- Cet outil vous posera un ensemble de questions, selon votre besoin répondez y et il fera tout le reste.

UserProvider

Le User Provider est un ensemble de classe associé au bundle Security de Symfony et qui ont deux rôles

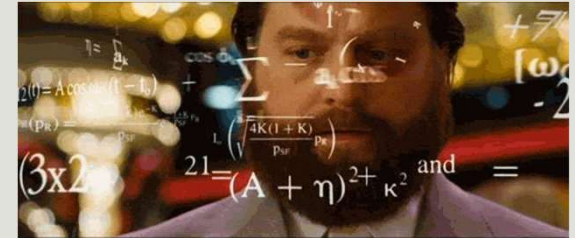
- **Récupérer le user de la session.** En effet, pour chaque requête, Symfony charge l'objet user de la session. Il vérifie aussi que le user n'a pas changé au niveau de la BD.
- Charge l'utilisateur pour réaliser certaines fonctionnalités comme la fonctionnalité ***se souvenir de moi.***

UserProvider

- Afin de définir le **userProvider** que vous voulez utiliser passer par le fichier de configuration **security.yaml**

```
providers:
  users:
    entity:
      # the class of the entity that represents users
      class: 'App\Entity\User'
      # the property to query by - e.g. username, email, etc
      property: email
```

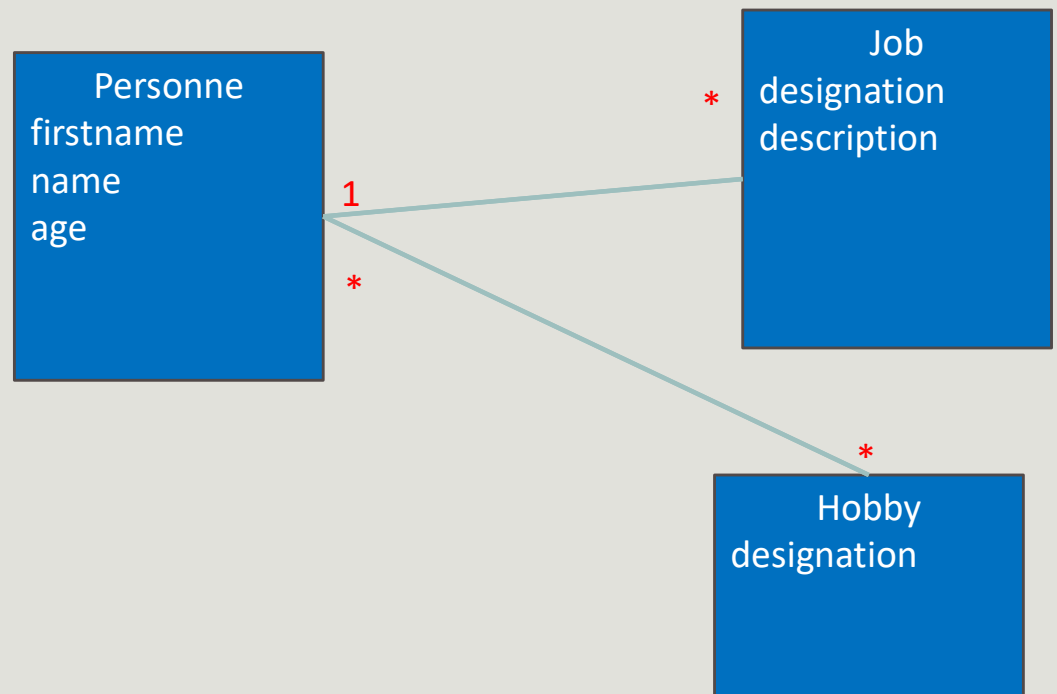

Exercice



Reprenez votre diagramme de classe.

Ajouter la classe user.

Ajouter les relations nécessaires.



Encoder le mot de passe

- Vous n'avez pas toujours besoin de mot de passe
- En cas de besoin, vous pouvez configurer la manière avec lequel votre mot de passe doit être géré dans le fichier security.yml.

```
# config/packages/security.yml
security:
  # ...
  password_hashers:
    Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
    # use your user class name here
    App\Entity\User:
      # Use native password hasher, which auto-selects the best
      # possible hashing algorithm (starting from Symfony 5.3 this is
      # "bcrypt")
      algorithm: auto
```

A partir de Symfony 5,3

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

Avant Symfony 5,3

<https://symfony.com/doc/6.4/security/passwords.html>

Encoder le mot de passe

- Le service responsable de l'encodage des mots de passe est le service `UserPasswordEncoder` (avant Symfony 5.3) ou `UserPasswordHasher` à partir de Symfony 5.3 .
- Afin de l'utiliser, et comme tous les services de Symfony, il suffit de l'injecter.

```
private $userPasswordHasher;  
public function __construct( UserPasswordEncoderInterface $userPasswordHasher)  
{  
    $this->userPasswordHasher = $userPasswordHasher;  
}
```

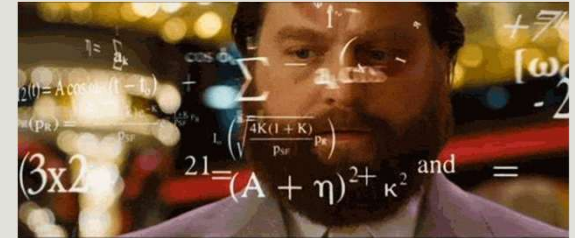
```
public function __construct( private UserPasswordEncoderInterface $passwordEncoder)  
{  
    $admin1->setPassword($this->hasher->hashPassword($admin1,'admin'))  
}
```

PHP 8

```
use Doctrine\Bundle\FixturesBundle\FixtureGroupInterface;
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;

class UserFixture extends Fixture implements FixtureGroupInterface
{
    public function __construct( private UserPasswordHasherInterface $passwordEncoder){}
    public function load(ObjectManager $manager, ): void
    {
        $user = new User();
        $user->setEmail('aymen@gmail.com')
            ->setPassword($this->passwordEncoder->hashPassword($user,'user'))
            ->setRoles(['ROLE_USER']);
        // ....
    }
    public static function getGroups(): array {
        return ['userGroup'];
    }
}
```

Exercice



- Créer un fixture qui permet de créer quelques utilisateurs.
- **Ne lancer que les fixtures du user.** Pour se faire, les fixtures que vous voulez lancer doivent implémenter l'interface `FixtureGroupeInterface`. Ceci permettra de lancer les fixtures d'un groupe donnée
- Utiliser `use Doctrine\Bundle\FixturesBundle\FixtureGroupInterface;`
- En implémentant cette interface, vous devez implémenter la méthode `getGroupes`. Cette méthode retourne un tableau contenant le nom des groupes auxquels appartient cette fixture.
- Exemple : `return ['groupeRedondant','groupeTest1'];`
- Enfin lancer la commande de chargement de fixture avec l'option `–group=nomGroupe` ajouter aussi l'option `–append` pour ne pas purger la base de données.

Authentication et Firewall

- Le système d'authentification de Symfony est configuré au niveau de la partie **firewalls** de votre fichier security.yaml.
- Cette section va définir comment vos utilisateurs seront authentifié, e.g. API Token, Formulaire d'authentification.

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

Authentication et Firewall

- Comme décrite dans la documentation, **l'authentification dans Symfony** ressemble à de « **la magie** ».
- En effet, au lieu d'aller construire une route et un contrôleur afin d'effectuer le traitement, vous devez simplement activer un « **authentication provider** ».
- « **L'authentication provider** » est du code qui s'exécute **automatiquement avant chaque appel d'un contrôleur**.
- Symfony possède un ensemble d'« authentication provider » prêt à l'emploi. Vous trouverez leur description dans la documentation :
https://symfony.com/doc/current/security/auth_providers.html
- Dans la documentation, il est conseillé de passer par les « Guard Authenticator » qui permettent un contrôle total sur toutes les parties de l'authentification.

Les Guard Authenticator

- Un « Guard authenticator » est une classe qui vous permet un control complet sur le processus d'authentification.
- Cette classe devra ou implémenter l'interface [AuthenticatorInterface](#) ou étendre la classe abstraite associée au besoin, e.g. [AbstractFormLoginAuthenticator](#) en cas de formulaire d'authentification ou [AbstractGuardAuthenticator](#) en cas d'api
- La commande **make:security** permet de générer automatiquement cette classe, il faut lui spécifier le type de l'authenticator.
- symfony console **make:security:form-login**

Redirection après le succès

- Vous pouvez personnaliser la redirection après le succès à travers l'attribut **default_target_path** du **form_login** dans **security.yaml**, passez-y la route qui se chargera de la redirection et implémentez-y votre logique métier

```
form_login:
  login_path: app_login
  check_path: app_login
  enable_csrf: true
  default_target_path: after_login
```

```
#[Route(path: '/loggedIn', name: 'after_login')]
public function afterLogin(): Response
{
    return $this->redirectToRoute('app_annonce_index');
}
```

Redirection après le logout

- Vous pouvez personnaliser la redirection après le logout à travers l'attribut **target de la clé logout dans security.yaml**, passez-y la route vers laquelle vous voulez rediriger votre user.

```
logout:  
    path: app_logout  
    target: app_login
```

Se déconnecter

- Afin de se **déconnecter**, il suffit d'ajouter la clé **logout** dans votre **firewalls configuration dans security.yaml**.
- Ajouter ensuite une **méthode vide logout dans votre securityController avec la route associé à votre méthode logout**.
- Vous pouvez débiter les autres options de logout avec la commande

symfony console debug:config security

```
/**
 * @Route("/logout", name="logout")
 */
public function logout() {

}
```

```
firewalls:
  main:
    logout:
      path: logout
```

Ajouter sa propre logique de vérification

- Lors de l'authentification d'un utilisateur, **des vérifications supplémentaires peuvent être nécessaires pour vérifier si l'utilisateur identifié est autorisé** à se connecter.
- En définissant un vérificateur d'utilisateur personnalisé, vous pouvez **définir dans votre pare-feu quel vérificateur doit être utilisé**.
- Les vérificateurs d'utilisateurs sont des **classes qui doivent implémenter l'interface `UserCheckerInterface`**.
- Cette interface définit deux méthodes, **`checkPreAuth()`** et **`checkPostAuth()`**, pour effectuer des vérifications avant et après l'authentification de l'utilisateur.
- Si **une ou plusieurs conditions ne sont pas remplies**, une **exception est générée, ce qui étend la classe `AccountStatusException`**.
- Pensez à utiliser **`CustomUserMessageAccountStatusException`**, qui étend **`AccountStatusException`** et permet de **personnaliser le message d'erreur** affiché à l'utilisateur.

```

namespace App\Security;

use App\Entity\User as AppUser;
use Symfony\Component\Security\Core\Exception\AccountExpiredException;
use Symfony\Component\Security\Core\Exception\CustomUserMessageAccountStatusException;
use Symfony\Component\Security\Core\User\UserCheckerInterface;
use Symfony\Component\Security\Core\User\UserInterface;

class UserChecker implements UserCheckerInterface
{
    public function checkPreAuth(UserInterface $user): void
    {
        if (!$user instanceof AppUser) {
            return;
        }

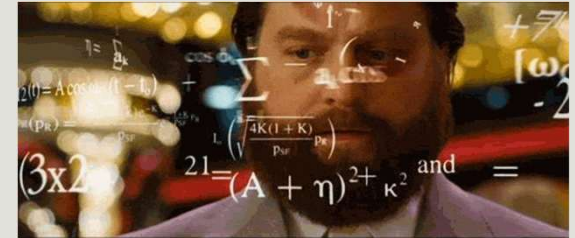
        if ($user->isDeleted()) {
            // the message passed to this exception is meant to be displayed to the user
            throw new CustomUserMessageAccountStatusException('Your user account no longer exists.');
```

Activer le vérificateur (checker)

➤ Afin **d'activer les vérificateurs** que vous avez définies, allez dans le firewall auquel vous souhaitez ajouter le vérificateur et avec la clé **user_checker** spécifier le **path** de votre checker :

```
main:
  lazy: true
  provider: app_user_provider
  user_checker: ~App\Security\ValidUserChecker
```

Exercice



- Créer un LoginForm en utilisant la commande
symfony console `make:security:form-login`
- Terminer les étapes définies par la commande à la fin de son exécution.

Reset Password

- Afin d'ajouter la fonctionnalité reset password vous pouvez utiliser le bundle reset-password-bundle :

composer require symfonycasts/reset-password-bundle

- Lancer ensuite la commande **symfony console make:reset-password**
- Suivez ensuite les étapes

Récupérer le user dans le contrôleur

Afin de récupérer le user dans un contrôleur, il suffit d'utiliser la méthode helper `getUser`.

Utiliser ensuite sa méthode

```
public function list()  
{  
    $user = $this->getUser();  
}
```

Récupérer le user dans le service

Afin de récupérer le user dans un service, il suffit d'injecter le Security Service.

Utiliser ensuite sa méthode `getUser`

```
use Symfony\Component\Security\Core\Security;
class HelperService
{
    private $security;
    public function __construct(Security $security)
    {
        $this->security = $security;
    }
    public function sendMoney() {
        $user = $this->security->getUser()
    }
}
```

Remember Me

- Afin d'activer la fonctionnalité 'se souvenir de moi' ajouter la configuration suivante dans le fichier `security.yml` :

```
# config/packages/security.yaml
security:
    # ...
    firewalls:
        main:
            # ...
            remember_me:
                secret: '%kernel.secret%' # required
                lifetime: 604800 # 1 week in seconds
```

- Décommenter le code affichant **le bouton remember** me dans la Twig `login.html.twig`

```
<div class="checkbox mb-3">
    <label>
        <input type="checkbox" name="_remember_me"> Remember me
    </label>
</div>
```

Register

- Afin de permettre l'ajout ou l'enregistrement de vos utilisateurs, vous pouvez utiliser la commande :

`symfony console make:registration-form`

- Cette fonctionnalité n'a rien de particulier, elle permet tout simplement d'ajouter un utilisateur dans votre base de données.
- Vous pouvez personnaliser le contrôleur généré comme vous le voulez.
- Si vous souhaitez activer la validation d'un utilisateur pour confirmer le mail et **avant de lancer le registration form installer le bundle via la commande**

`composer require symfonycasts/verify-email-bundle`

Authentification manuelle d'un utilisateur

- Une fois l'utilisateur inscrit, vous pourrez l'authentifier d'une façon manuelle en injectant le service **UserAuthenticatorInterface**.
- Utiliser sa méthode authenticateUser qui prend en paramètre le user, votre authenticator et l'objet request

```
public function register(Request $request, UserPasswordHasherInterface $userPasswordHasherInterface,
LoginFormAuthenticator $authenticator,
UserAuthenticatorInterface $userAuthenticator): Response
{
    //...
    if ($form->isSubmitted() && $form->isValid()) {
        //...
        // Authenticate user
        // retourne un Objet Response, celui généré par la méthode onAuthenticationSuccess
        return $userAuthenticator->authenticateUser($user, $authenticator, $request);
    }
}
```

Autorisation

- Processus permettant d'autoriser un utilisateur à accéder à une ressource selon son rôle.

Le processus d'authentification suit deux étapes.

- 1- Lors de l'authentification l'utilisateur est associé à un ensemble de rôles.
- 2- Lors de l'accès à une ressource, on vérifie si l'utilisateur a le rôle nécessaire pour y accéder.

Les Rôles

- Chaque utilisateur connecté a au moins un rôle : le `ROLE_USER`
- Tous les rôles commencent par `ROLE_`

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

    return array_unique($roles);
}
```

Définir les droits d'accès

Les droits d'accès sont définis de deux façons :

- 1- Dans le fichier `security.yaml`
- 2- Directement dans la ressource

Sécuriser les patrons d'url

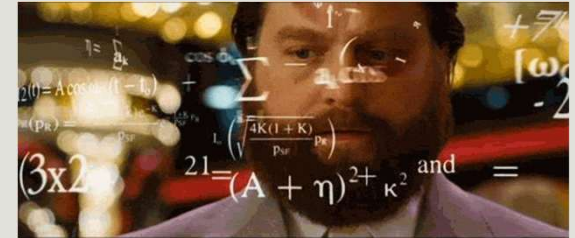
- La méthode la plus basique pour sécuriser une partie de votre application est de sécuriser un patron d'url complet dans votre fichier security.yaml.
- Ceci se fait sous la clé access_control. Chaque entrée est un objet avec comme clé :
 - **-path** : le pattern à protéger
 - **-roles** : les rôles qui peuvent accéder à ce pattern.
- Lorsque vous essayer d'accéder à une ressource, Symfony cherche dans cette rubrique s'il y a un matching avec la route recherchée de haut vers le bas. Le premier qu'il trouve lui permet de vérifier si vous avez le bon rôle pour accéder à la ressource demandée ou non. **L'ordre donc est très important.**

access_control:

- { **path**: ^/admin, **roles**: ROLE_ADMIN }
- { **path**: ^/profile, **roles**: ROLE_USER }

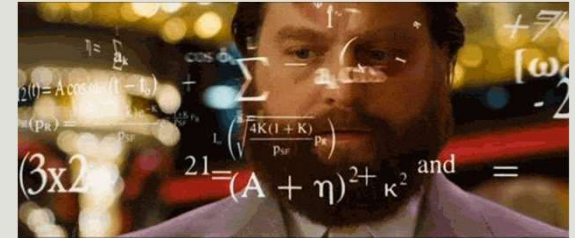
https://symfony.com/doc/current/security/access_control.html

Exercice



- Créer une action avec la route `'/admin'`. Décommenter les **access_control** et essayer deux scénarios.
 1. Connecter vous en tant que USER et essayer d'accéder à cette route. Que se passe t-il ?
 2. Déconnecter vous et essayer d'y accéder. Que se passe t-il ?
- Modifier votre classe UserFixture et faite en sorte d'ajouter un user avec le `ROLE_ADMIN`. N'effacer rien de votre base.
- Connecter vous en tant qu'admin. Essayer d'accéder à la route `/admin`. Que se passe t-il ?

Exercice



- Créer une action permettant d'inscrire des utilisateurs.
- Créer une action pour l'admin lui permettant d'ajouter des utilisateurs avec le ROLE qu'il veut.

Définir la route à activer en cas d'erreur 401 ou 403

- Dans Symfony, vous pouvez lancer une exception **AccessDeniedException** pour interdire l'accès à l'utilisateur. Symfony **gère cette exception et génère une réponse en fonction de l'état d'authentification** :
- Si **l'utilisateur n'est pas authentifié (ou authentifié anonymement)**, un point d'entrée d'authentification est utilisé pour générer une réponse (**généralement une redirection vers la page de connexion ou une réponse 401 « Non autorisé »**) ;
- Si l'utilisateur est authentifié, mais ne dispose pas des autorisations requises, une réponse 403 « Interdit » est générée.
- Vous pouvez personnaliser la gestion de l'erreur 401 et 403.

Définir la route à activer en cas d'erreur 401

- Pour l'erreur 401, vous devez créer une classe qui implémente `AuthenticationEntryPointInterface`. Cette interface possède une méthode `start()` appelée chaque fois qu'un utilisateur non authentifié tente d'accéder à une ressource protégée.
- Une fois implémenté configurer votre firewall afin qu'il utilise cette classe via la clé **entry_point**

```
main:
    lazy: true
    provider: app_user_provider
    entry_point: App\Security\AuthenticationEntryPoint
    custom_authenticator: App\Security\AuthAuthenticator
```

Définir la route à activer en cas d'erreur 403

- Pour l'erreur **403**, créez une classe implémentant **AccessDeniedHandlerInterface**. Cette interface définit une méthode appelée **handle()** permettant d'implémenter la logique à exécuter lorsque l'accès est refusé à l'utilisateur actuel (par exemple, envoyer un e-mail, consigner un message ou renvoyer une réponse personnalisée).
- Une fois implémenté configurer votre **firewall** afin qu'il utilise cette classe via la clé **access_denied_handler**

```
firewalls:
  dev:
    pattern: ^/(_(profiler|wdt)|css|images|js)/
    security: false
  main:
    lazy: true
    provider: app_user_provider
    access_denied_handler: App\Security\AccessDeniedHandler
    custom_authenticator: App\Security\AuthAuthenticator
```

Sécuriser Les contrôleurs

Vous pouvez directement sécuriser vos contrôleurs en utilisant :

- 1- Le helper **denyAccessUnlessGranted**('ROLE_*');
- 2- En utilisant l'annotation **@IsGranted**('ROLE_*')

```
/**
 * Matches /blog exactly
 *
 * @IsGranted("ROLE_ADMIN")
 * @Route("/blog", name="blog_list")
 */
public function list()
{
    // ...
}
```

```
/**
 * Matches /blog exactly
 *
 * @Route("/blog", name="blog_list")
 */
public function list()
{
    $this->denyAccessUnlessGranted("ROLE_USER");
    // ...
}
```

Sécuriser un service

Afin de sécuriser un service, il suffit d'injecter le Security Service.

Utiliser ensuite sa méthode **isGranted**

```
use Symfony\Component\Security\Core\Security;
class HelperService
{
    private $security;
    public function __construct(Security $security)
    {
        $this->security = $security;
    }
    public function sendMoney() {
        if ($this->security->isGranted("ROLE_ADMIN")) {
            // Todo Send Money
        }
    }
}
```

https://symfony.com/doc/current/security/securing_services.html

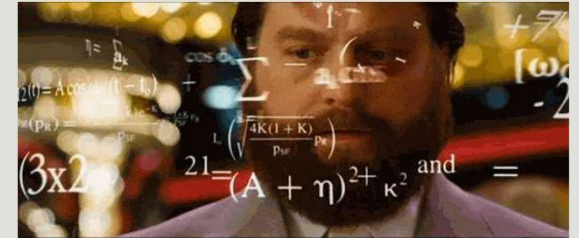
Sécuriser vos pages TWIG

Si vous voulez vérifier le rôle de l'utilisateur avant d'afficher une ressource ou des informations dans vos pages TWIG, utiliser la méthode `is_granted('ROLE_*)`

```
{% if is_granted('ROLE_ADMIN') %}  
    <a href=« /admin">Administration</a>  
{% endif %}
```

Exercice

Faite en sorte que le menu s'adapte au rôle de l'utilisateur connecté.



Hiérarchie de rôles

- Vous pouvez définir une hiérarchie de rôles.
- Dans le fichier **security.yaml** et sous la clé **role_hierarchy**, définissez le **rôle principal** suivie de **l'ensemble des rôles dont il hérite**.
- Un use case très récurant est quand l'admin possède tous les droits, donc l'admin devra hériter de tous les rôles.

```
role_hierarchy:  
  ROLE_COMMERCIAL:      ROLE_AGENT  
  ROLE_SECRETARY:       ROLE_COMMERCIAL  
  ROLE_ADMIN:           [ROLE_PARTNER, ROLE_SECRETARY]
```

- Ici un commercial a les accès de l'Agent.
- L'admin peut avoir les accès du Partner et de la secrétaire.

Spécial Rôles

- **PUBLIC_ACCESS** : Un utilisateur non authentifié
- **IS_AUTHENTICATED_REMEMBERED** : Vérifie qu'un utilisateur est authentifié indépendamment de son ROLE.
- **IS_AUTHENTICATED_FULLY** : Vérifie qu'un utilisateur est authentifié indépendamment de son ROLE. Cependant si le user est authentifié à cause de la fonctionnalité 'remember_me' alors il n'est pas authenticated fully.