


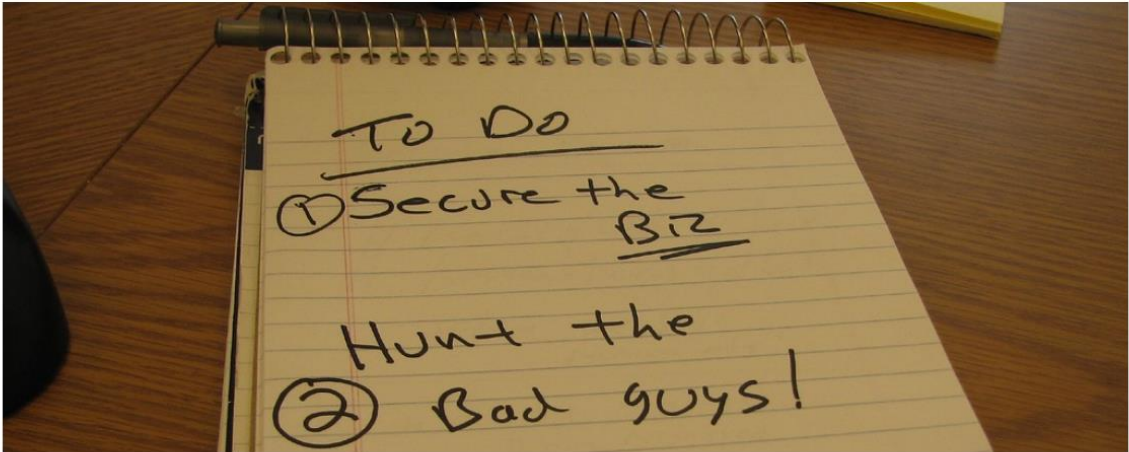
Implémentation de l'authentification

➤ Point de vue de l'utilisateur

Un utilisateur non authentifié est automatiquement redirigé sur la page de connexion :

To Do List app






Nom d'utilisateur :

Mot de passe :

Se connecter

L'utilisateur s'authentifie à l'aide d'un formulaire de connexion. Après avoir renseigné son nom d'utilisateur et son mot de passe, il clique sur le bouton « Se connecter ». Si les identifiants renseignés sont valides, l'utilisateur est désormais authentifié et il est redirigé sur la page de gestion des tâches :

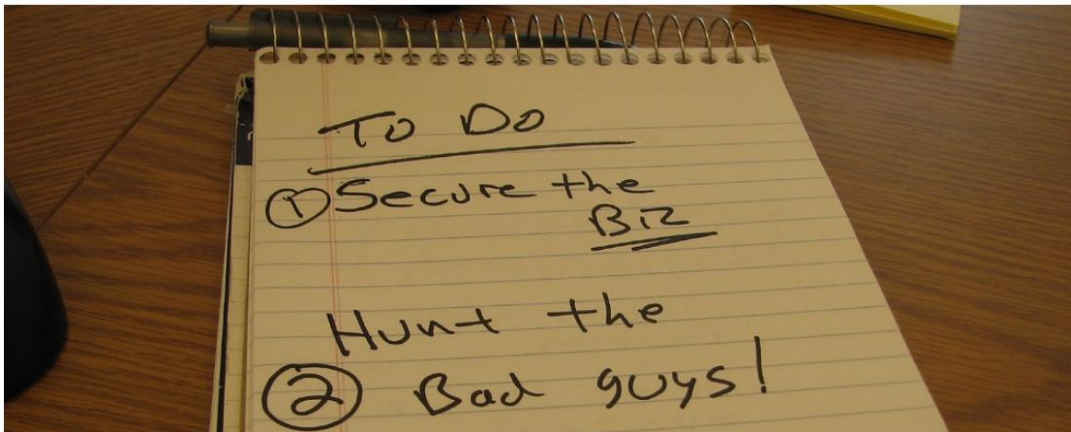
To Do List app



Se déconnecter

Bienvenue sur Todo List,

l'application vous permettant de gérer l'ensemble de vos tâches sans effort !



Créer une nouvelle tâche

Consulter la liste de toutes les tâches

Consulter la liste des tâches à faire

Consulter la liste des tâches terminées

➤ Préambule

La protection d'une application passe par la protection des URL, et donc des routes associées à ces URL. Les routes sont définies par les méthodes dans les contrôleurs (classes définies dans le répertoire src/Controller) :

- Une partie des routes peut être accessible au grand public [visiteurs non authentifiés]
- Une autre partie des routes peut être accessible seulement à certains visiteurs authentifiés.

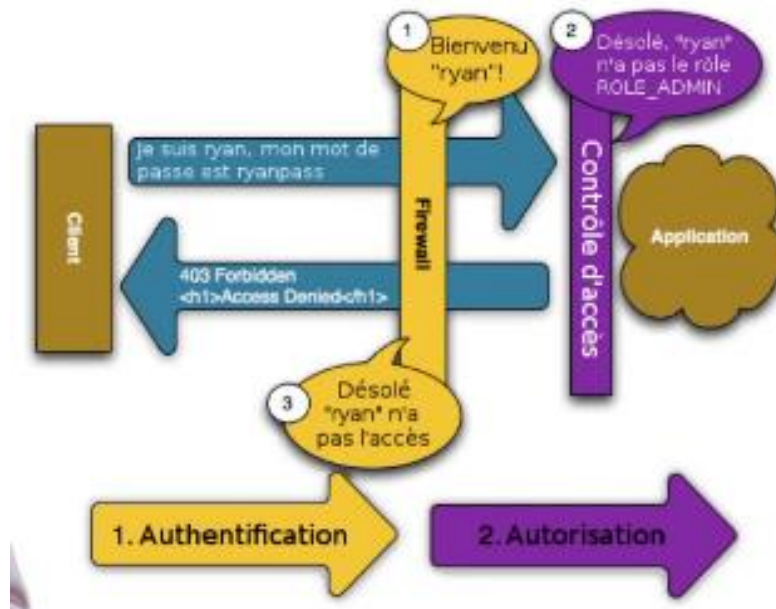
Donc, en matière de sécurité, on parle de deux notions différentes :

- 1- **L'authentification** : qui identifie le visiteur,
- 2- **L'autorisation** : qui permet de savoir si le visiteur authentifié a les droits d'accès à une fonctionnalité ou à une page



Ces deux notions sont consécutives : **d'abord** le visiteur est authentifié, **puis** l'autorisation détermine les droits d'accès à telle ou telle URL.

Par exemple, voici ci-après un schéma qui explique qu'un utilisateur authentifié n'a pas les droits pour accéder à une page qui requiert le rôle « `ROLE_ADMIN` ».



Images tirées du cours sur OpenClassrooms « Développer votre site avec le framework Symfony 3 » de Fabien Potencier, créateur de Symfony

Au final, voici **les étapes** pour un utilisateur qui tente **d'accéder** à une **ressource protégée** :

1. Un visiteur veut accéder à une ressource protégée.
2. Il est renvoyé vers un formulaire d'identification par le **firewall**.
3. L'utilisateur soumet ses informations d'identification (ex : login et mot de passe)
4. Le **firewall** authentifie l'utilisateur en confirmant son identification.
5. La requête de départ est renvoyée par le visiteur authentifié.
6. L'**access-control** vérifie les droits de l'utilisateur, et autorise ou non l'accès à la ressource protégée.

Le processus de sécurité suit toujours ces étapes mais la nature de l'authentification peut varier. En effet, selon vos besoins, il existe de nombreuses manières d'authentifier les visiteurs (banal formulaire d'identification, authentification avec Google, etc). Il faut retenir que la méthode choisie n'a pas d'incidence sur votre code dans les contrôleurs.

Dans ce document, nous allons expliquer l'implémentation de **l'authentification de l'application**, qui concerne les points 1 à 4 des étapes énumérées ci-avant.

NB : Pour en savoir plus sur **l'autorisation** sous Symfony, vous pouvez consulter la [documentation officielle de Symfony](#), et un [cours sur OpenClassrooms de Mickaël Andrieu](#)

➤ L'authentification dans Symfony

L'authentification est le procédé qui permet de **déterminer qui est votre visiteur**.

Il y a deux cas possible :

- Soit le visiteur est **anonyme** car il n'est pas identifié,
- Soit le visiteur est **membre** car il est identifié [via un formulaire ou via un cookie].

La procédure d'authentification va donc déterminer si le visiteur est anonyme ou membre du site.

C'est le **firewall** (ou « pare-feu ») qui gère l'authentification sous Symfony.

Ainsi, on peut restreindre l'accès à certaines parties du site uniquement aux visiteurs qui sont membres. Autrement dit, il faudra que le visiteur soit authentifié pour que le firewall l'autorise à passer.

Par exemple, dans l'application ToDo & Co, seuls les visiteurs authentifiés (membres) ont accès aux pages de gestion des tâches (pages de liste des tâches, modifier une tâche, etc).

NB : Après l'authentification, intervient l'autorisation qui gère les droits d'accès des différents visiteurs membres. Sous Symfony, l'autorisation est prise en charge par **l'access-control**.

Par exemple, dans l'application ToDo & Co, c'est l'autorisation qui gère le fait que seul un visiteur avec le rôle `ROLE_ADMIN` peut accéder aux pages de gestion des utilisateurs.

➤ Implémentation de l'authentification

1- Installation de la fonction de sécurité de Symfony

Pour gérer la sécurité dans Symfony, on va d'abord s'assurer de la présence du composant de sécurité : **symfony/security-bundle**.

Dans les applications utilisant [Symfony Flex](#), on exécute la commande ci-après pour installer la fonction de sécurité avant de l'utiliser :

```
composer require symfony/security-bundle
```

2- Création de la classe User :

Peu importe *comment* vous vous authentifierez (par exemple, formulaire de connexion ou token API) ou *où* vos données utilisateur seront stockées (base de données, authentification unique), l'étape suivante est toujours la même : créer une classe « **User** ».

Une seule contrainte pour la création d'une classe utilisateur est d'implémenter l'interface `Symfony\Component\Security\Core\User\UserInterface` du composant **Security**.

Dans l'application **ToDo & Co**, les **utilisateurs** sont **stockés** en **base de données**. Donc ici, la classe **User** est une **entité** qui permet de stocker les données des utilisateurs en base de données.

Voici le lien vers la classe entité **User** contenue dans le répertoire **src/Entity/** :

<https://github.com/CarolineDirat/P8ImproveToDo/blob/master/src/Entity/User.php>

Remarquez que l'on utilise [Doctrine ORM](#) (`Doctrine\ORM\Mapping` as **ORM**) pour persister les utilisateurs :

- la classe User est notifiée comme entité par l'annotation **@ORM\Entity** pour correspondre à la table « user » dans la base de données ;
- et les propriétés de la classe User sont notifiées par l'annotation **@ORM\Column** pour pouvoir faire correspondre à chacune un champ dans la table user, dans la base de données.

NB : L'annotation **@Assert** (alias de `Symfony\Component\Validator\Constraints`) permet de définir les [contraintes de validation](#) vérifiées lors de la création d'un utilisateur, et ne concerne donc pas l'authentification.

NB : On peut facilement créer un utilisateur grâce au [MakerBundle](#), avec la commande :

```
php bin/console make:user
```

On peut alors préciser le nom de la classe utilisateur, si l'utilisateur sera stocké en base de données, la propriété qui correspondra au login et si le mot de passe sera hashé :

```
The name of the security user class (e.g. User) [User]:
```

```
> User
```

```
Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
```

```
> yes
```

```
Enter a property name that will be the unique "display" name for the user (e.g.
email, username, uuid [email])
> email
```

```
Does this app need to hash/check user passwords? (yes/no) [yes]:
> yes
```

```
created: src/Entity/User.php
created: src/Repository/UserRepository.php
updated: src/Entity/User.php
updated: config/packages/security.yaml
```

Si la classe User est une entité (comme dans notre cas), vous pouvez utiliser la [commande make:entity](#) pour ajouter plus de champs :

```
php bin/console make:entity
```

Assurez-vous également d'effectuer et d'exécuter une migration pour la nouvelle entité :

```
php bin/console make:migration
php bin/console doctrine:migrations:migrate
```

3- Configuration de l'authentification

Un fichier de configuration est dédié à la sécurité, c'est le fichier **security.yaml** qui se trouve dans le répertoire **config/packages** de l'application Symfony.

Voici le fichier security.yaml de notre application :

```
# config/packages/security.yaml
security:
    encoders:
        # use your user class name here
        App\Entity\User:
            # Use native password encoder
            # This value auto-selects the best possible hashing algorithm
            # (i.e. Sodium when available).
            algorithm: 'auto'
    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: 'username'
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt|error)|css|images|js)/
            security: false
        main:
            anonymous: lazy
            provider: users
            form_login:
                login_path: login
                check_path: login
                always_use_default_target_path: true
                default_target_path: /
                csrf_token_generator: security.csrf.token_manager
            logout:
                path: logout
                target: login
```

```
# 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: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
  - { path: ^/users, roles: ROLE_ADMIN }
  - { path: ^/, roles: ROLE_USER }
```

L'implémentation de l'authentification passe donc maintenant par la définition de certaines parties de ce fichier de configuration : **security.providers**, **security.encoders** et **security.firewalls**.

a- Le « User Provider » ou « Fournisseur d'utilisateur » : **security.providers**

```
# config/packages/security.yaml
security:
  # ...
  # https://symfony.com/doc/current/security.html#where-do-users-come-from-user-providers
  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: 'username'
  # ...
```

Pour identifier les utilisateurs, le firewall demande à un **provider**. Le provider est un **fournisseur d'utilisateurs**.

Dans notre cas, les utilisateurs sont stockés en base de données, via l'entité User. Le Provider « **users** » est donc défini pour nous permettre de récupérer les utilisateurs depuis la base de données. Le provider « **users** » doit connaître :

- La classe de l'entité User qui représente les utilisateurs :
providers.users.entity.class: 'App\Entity\User'
- La propriété de la classe User qui sert d'identifiant (login)
providers.users.entity.property: 'username'

b- Le codage du mot de passe : **security.encoders**

```
# config/packages/security.yaml
security:
  # ...
  encoders:
    # use your user class name here
    App\Entity\User:
      # Use native password encoder
      # This value auto-selects the best possible hashing algorithm
      # (i.e. Sodium when available).
      algorithm: 'auto'
  # ...
```

Ici on configure comment encoder le mot de passe, qui ne peut pas être enregistré tel quel en base de données pour des raisons évidentes de sécurité.

Maintenant que Symfony sait *comment* vous voulez encoder les mots de passe, vous pouvez utiliser le service **UserPasswordEncoderInterface** pour encoder le mot de passe avant d'enregistrer vos utilisateurs dans la base de données.

NB : Dans **ToDo & Co**, c'est un **entity listener** « [UserPasswordListener](#) » qui s'occupe d'encoder le mot de passe via le service **UserPasswordEncoderInterface**, juste avant la première persistance de l'utilisateur en base de données (lors de la création d'un utilisateur donc). Pour cela, l'[entity listener](#) écoute l'[événement doctrine](#) `prePersist`, comme c'est indiqué dans sa déclaration dans le fichier [config/service.yaml](#).

c- Authentification et Firewall (pare-feu) – quelques explications

security.firewalls

Un **pare-feu** (firewall) est votre système d'authentification : il définit *comment* vos utilisateurs pourront s'authentifier (par exemple, formulaire de connexion, token API, etc.).

Un seul pare-feu est actif à chaque requête : Symfony utilise la clé **pattern** pour trouver la première correspondance (vous pouvez également faire [correspondre par hôte ou autre chose](#)).

```
# config/packages/security.yaml
security:
  # ...
  firewalls:
    dev:
      pattern: ^/(_(profiler|wdt|error)|css|images|js)/
      security: false
    main:
      anonymous: lazy
      provider: users
      form_login:
        login_path: login
        check_path: login
        always_use_default_target_path: true
        default_target_path: /
        csrf_token_generator: security.csrf.token_manager
      logout:
        path: logout
        target: login
  # ...
```

Le pare-feu « **dev** » est un faux pare-feu : il garantit que vous ne bloquiez pas accidentellement les outils de développement de Symfony - qui sont sous des URL comme `/_profiler` et `/_wdt` :

Le pare-feu « **main** » configure la méthode d'authentification de l'application.

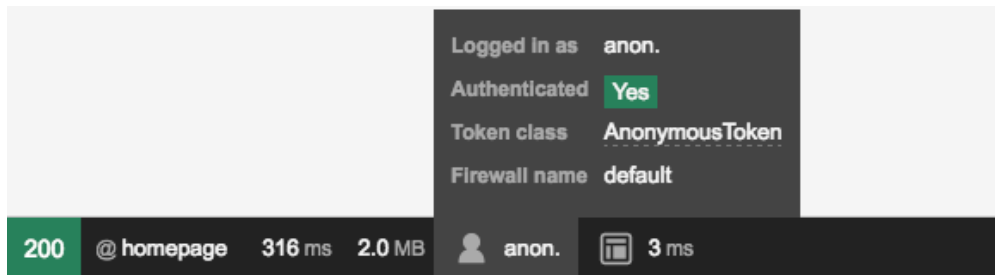
Toutes les URL *réelles* sont gérées par le pare-feu « **main** » :

- **Aucune clé pattern** signifie qu'elle correspond à **toutes les URL**.
- On indique que le **fournisseur d'utilisateur** à utiliser est « **users** »
(Avec « **provider: users** »)

Un pare-feu peut avoir de nombreux modes d'authentification, c'est-à-dire de nombreuses façons de poser la question « Qui êtes-vous ? ».

Souvent, lors de sa première visite sur votre site Web, l'utilisateur est inconnu (c'est-à-dire non connecté). Le mode **anonymous**, s'il est activé, est utilisé pour ces demandes.

En fait, si vous allez maintenant sur la page d'accueil en mode dev, vous y aurez accès et vous verrez que vous êtes « authentifié » en tant que **anon.**. Le pare-feu a vérifié qu'il ne connaît pas votre identité et que vous êtes donc anonyme :



Cela signifie que toute demande peut avoir un **token anonyme** pour accéder à certaines ressources, tandis que certaines actions (c'est-à-dire certaines pages ou boutons) peuvent encore nécessiter des privilèges spécifiques. Un utilisateur peut alors accéder à un formulaire de connexion sans être authentifié en tant qu'utilisateur unique (sinon une boucle de redirection infinie se produirait en demandant à l'utilisateur de s'authentifier tout en essayant de le faire).

Puis c'est le système d'autorisation qui permettra de refuser l'accès à certaines URL, contrôleurs ou partie de modèles.

NB : Le mode **anonymous: lazy** empêche le démarrage de la session s'il n'y a pas besoin d'autorisation. Ceci est important pour garder les requêtes en cache (voir [Cache HTTP](#)).

NB : Si vous ne voyez pas la barre d'outils, installez le [profileur](#) avec:

```
composer require --dev symfony/profiler-pack
```

Maintenant que nous comprenons notre pare-feu, l'étape suivante consiste à créer un moyen pour nos utilisateurs de s'authentifier !

d- Méthode d'authentification d'un utilisateur configurée dans le pare-feu

L'authentification dans Symfony peut sembler un peu « magique » au début. En effet, au lieu de créer une route et un contrôleur pour gérer la connexion, on active **un fournisseur d'authentification** : du code qui s'exécute automatiquement *avant que* votre contrôleur ne soit appelé.

Symfony dispose de plusieurs [fournisseurs d'authentification intégrés](#). L'application ToDo & Co utilise le [fournisseur d'authentification form login](#), qui gère automatiquement un formulaire de connexion POST. Il est défini dans le firewall « **main** » :


```
# config/packages/security.yaml
security:
  # ...
  firewalls:
    # ...
    main:
      anonymous: lazy
      provider: users
      form_login:
        login_path: login
        check_path: login
      # ...
```

→ Active le fournisseur d'authentification form_login

Désormais, lorsque le système de sécurité lance le processus d'authentification, il redirige l'utilisateur vers le formulaire de connexion sur l'URL « **/login** », qui sera géré dans [SecurityController](#) en définissant la route « **login** » (définie dans la configuration de form_login dans security.yaml) :

```
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;

class SecurityController extends AbstractController
{
    /**
     * login.
     *
     * @Route("/login", name="login", methods={"GET", "POST"})
     *
     * @param AuthenticationUtils $authenticationUtils
     *
     * @return Response
     */
    public function login(AuthenticationUtils $authenticationUtils): Response
    {
        // 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,
            ]
        );
    }
}
```

Ne laissez pas ce contrôleur vous confondre. Lorsque l'utilisateur soumet le formulaire, le système de sécurité gère automatiquement la soumission du formulaire pour vous. Si l'utilisateur soumet un nom d'utilisateur ou un mot de passe invalide, ce contrôleur lit

l'erreur de soumission de formulaire à partir du système de sécurité, afin qu'elle puisse être affichée à nouveau à l'utilisateur :

```
// get the login error if there is one
$error = $authenticationUtils->getLastAuthenticationError();
```

En d'autres termes, il ne reste plus qu'à *afficher* le formulaire de connexion et toutes les erreurs de connexion qui ont pu se produire :

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

Mais c'est le système de sécurité lui-même qui se charge de vérifier le nom d'utilisateur et le mot de passe soumis et d'authentifier l'utilisateur.

Donc, voici le **formulaire de connexion** de ToDo & Co :

```
{# /templates/security/login.html.twig #}
{% extends 'base.html.twig' %}

{% block body %}
    {% if error %}
        <div class="alert alert-
danger" role="alert">{{ error.messageKey|trans(error.messageData, 'security') }}</div>
    {% endif %}

    <form action="{{ path('login') }}" method="post" class="form-row">
        <div class="col-md-4 col-lg-3 mt-2">
            <label for="username">Nom d'utilisateur :</label>
            <input type="text" class="form-control" id="username" name="_username"
value="{{ last_username }}" />
        </div>
        <div class="col-md-4 col-lg-3 mt-2">
            <label for="password">Mot de passe :</label>
            <input type="password" class="form-control" id="password" name="_password" />
        </div>
        <div class="col d-flex align-items-end mt-2">
            → <input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}" />
            <button class="btn btn-success " type="submit">Se connecter</button>
        </div>
    </form>

{% endblock %}
```

L'aspect du formulaire peut varier, mais il suit généralement certaines conventions :

- L'élément **<form>** envoie une requête **POST** à la route « **login** », car c'est ce qui est configuré sous la clé d'entrée **form_login** dans `security.yaml`;
- Le champ du nom d'utilisateur a le nom **_username** et le champ du mot de passe a le nom **_password**.

Remarquez au niveau de la flèche rouge qu'un champ caché permet de générer un token CSRF grâce à la fonction **csrf_token()** de Twig.

Aussi, on configure le **fournisseur de token CSRF** utilisé par le formulaire de connexion dans la configuration de sécurité :

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

    firewalls:
        # ...
        main:
            # ...
            form_login:
                # ...
                csrf_token_generator: security.csrf.token_manager
```

On a défini l'utilisation du **fournisseur par défaut** disponible dans le composant de sécurité, donc le champ HTML du token csrf doit être appelé « **_csrf_token** » et la chaîne utilisée pour générer la valeur doit être « **authenticate** »:

```
<input type="hidden" name="_csrf_token" value="{{ csrf_token('authenticate') }}" />
```

Enfin, on précise **deux autres options** au fournisseur d'authentification **form_login** :

```
# config/packages/security.yaml
security:
    # ...
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt|error)|css|images|js)/
            security: false
        main:
            anonymous: lazy
            provider: users
            form_login:
                login_path: login
                check_path: login
                default_target_path: /
                always_use_default_target_path: true
                csrf_token_generator: security.csrf.token_manager
    # ...
```

L'option **default_target_path** permet de définir la page vers laquelle l'utilisateur est redirigé si aucune page précédente n'a été stockée dans la session. Ici c'est la page d'accueil à l'URL « / ».

L'option booléenne **always_use_default_target_path** permet d'ignorer l'URL demandée précédemment et toujours rediriger vers la page par défaut (ici, la page d'accueil à l'URL « / »).

Et voilà !

Lorsque vous soumettez le formulaire, le système de sécurité vérifie automatiquement les informations d'identification de l'utilisateur et authentifie l'utilisateur, ou en cas d'échec renvoie l'utilisateur au formulaire de connexion l'affichage de l'erreur identifiée (token CSRF invalide ou identifiants invalides).

Pour revoir l'ensemble du processus :

- 1- L'utilisateur tente d'accéder à une ressource protégée ;
- 2- Le pare-feu lance le processus d'authentification en redirigeant l'utilisateur vers le formulaire de connexion (/login);
- 3- La page /login rend le formulaire de connexion via la route « login » du SecurityController;
- 4- L'utilisateur soumet le formulaire de connexion de la page /login;
- 5- Le système de sécurité intercepte la demande, vérifie les informations d'identification soumises par l'utilisateur, authentifie l'utilisateur si elles sont correctes et renvoie l'utilisateur au formulaire de connexion si elles ne le sont pas.

Une fois l'utilisateur authentifié, il ne passera plus par le formulaire de connexion accéder aux URLs protégées par le pare-feu, mais l'access-control vérifiera ses droits à chaque requête.

4- La déconnexion

Pour activer la déconnexion, on **active** le paramètre de configuration **logout** sous le pare-feu « **main** » ; en précisant la route de déconnexion (**logout.path: logout**) :

```
#config/packages/security.yaml
security:
  # ...
  firewalls:
    # ...
    main:
      anonymous: lazy
      provider: users
      form_login:
        login_path: login
        check_path: login
        always_use_default_target_path: true
        default_target_path: /
        csrf_token_generator: security.csrf.token_manager
      → logout:
        path: logout
        target: login
```

Ensuite, on crée une route pour cette URL (et non pas dans le SecurityController)

```
# config/routes.yaml
logout:
  path: /logout
  methods: GET
```

Et c'est tout !

En envoyant un utilisateur vers la route **logout** (c'est-à-dire vers l'URL **/logout**), Symfony désauthentifie l'utilisateur actuel et le redirige vers l'URL **/login** (grâce à l'option **logout.target: login** dans security.yaml).

NB : Besoin de plus de contrôle sur ce qui se passe après la déconnexion ? Ajoutez une clé **success_handler** sous **logout** et pointez-la vers un identifiant de service d'une classe qui implémente `Symfony\Component\Security\Http\Logout\LogoutSuccessHandlerInterface`.

5- Accéder à l'objet User de l'utilisateur authentifié :

Quelle que soit la méthode d'authentification, on accède à l'objet User de l'utilisateur authentifié de la même façon :

- dans un contrôleur via la méthode **getUser()**.

```
public function index()
{
    // usually you'll want to make sure the user is authenticated first
    $this->denyAccessUnlessGranted('IS_AUTHENTICATED_FULLY');

    // returns your User object, or null if the user is not authenticated
    // use inline documentation to tell your editor your exact User class
    /** @var \App\Entity\User $user */
    $user = $this->getUser();

    // Call whatever methods you've added to your User class
    // For example, if you added a getFirstName() method, you can use that.
    return new Response('Well hi there '.$user->getFirstName());
}
```

- dans un service, via le service **Symfony\Component\Security\Core\Security**

```
// src/Service/ExampleService.php
// ...

use Symfony\Component\Security\Core\Security;

class ExampleService
{
    private $security;

    public function __construct(Security $security)
    {
        // Avoid calling getUser() in the constructor: auth may not
        // be complete yet. Instead, store the entire Security object.
        $this->security = $security;
    }

    public function someMethod()
    {
        // returns User object or null if not authenticated
        $user = $this->security->getUser();
    }
}
```

- Dans un modèle via la variable **app.user** grâce à la [variable d'application globale Twig](#) :

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

➤ Conclusion

Au final, les fichiers concernés par l'authentification sont :

- **config/packages/security.yaml** : le fichier qui configure la sécurité de l'application, L'authentification y est configurée sous les éléments :
 - **security.encoders** qui définit la méthode d'encodage du mot de passe.
 - **security.providers** qui définit le fournisseur d'utilisateurs.
 - **security.firewalls** qui configure la méthode d'authentification, et où on précise le fournisseur d'utilisateur (provider) à utiliser et les URL gérées par le pare-feu (pattern).
- **src/Controller/SecurityController.php** : le fichier qui implémente la route de connexion définie dans le pare-feu configuré dans le fichier config/packages/security.yaml.
- le formulaire de connexion dans **templates/security/login.html.twig**.
- et éventuellement, **config/routes.yaml** pour déclarer la route de déconnexion définie dans le pare-feu.

La méthode d'authentification configurée dans **security.firewalls** n'a pas d'incidence sur le code dans les contrôleurs. C'est-à-dire que l'on peut modifier celle-ci sans impacter sur le code exécuté dans les contrôleurs.

***Par exemple**, pour avoir un contrôle complet sur votre formulaire de connexion, Symfony recommande de créer une authentification de connexion par formulaire avec Guard. Comme il est expliqué dans la [documentation officielle](#) de Symfony, il s'agira de créer un **authentificateur Guard** nommé « LoginFormAuthenticator », héritant de `AbstractFormLoginAuthenticator` et implémentant `PasswordAuthenticatedInterface`. Et dans `security.yaml`, il faudra modifier **security.firewalls.main** pour y définir le fournisseur d'authentification **guard** à la place de **form_login**.*

Une fois la méthode d'authentification modifiée, vous n'aurez pas à modifier le code des contrôleurs, ni la façon d'accéder à l'utilisateur authentifié (dans un controller, un service ou un modèle).

Pour aller plus loin, vous avez la [documentation officielle de Symfony](#).