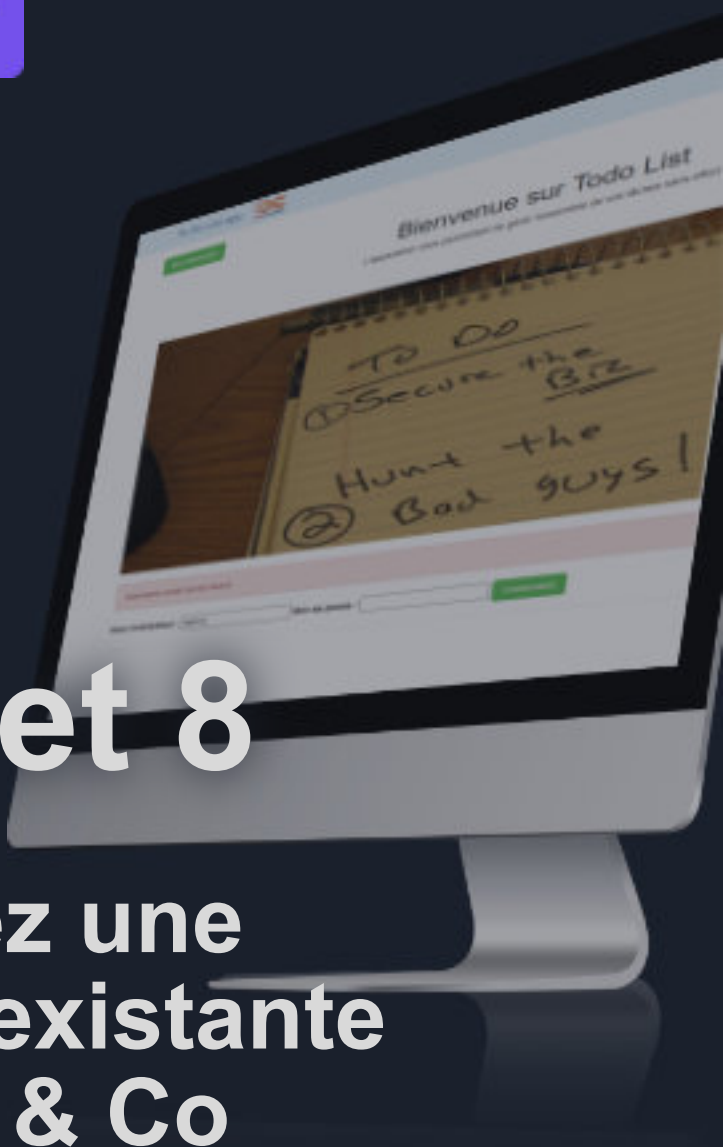
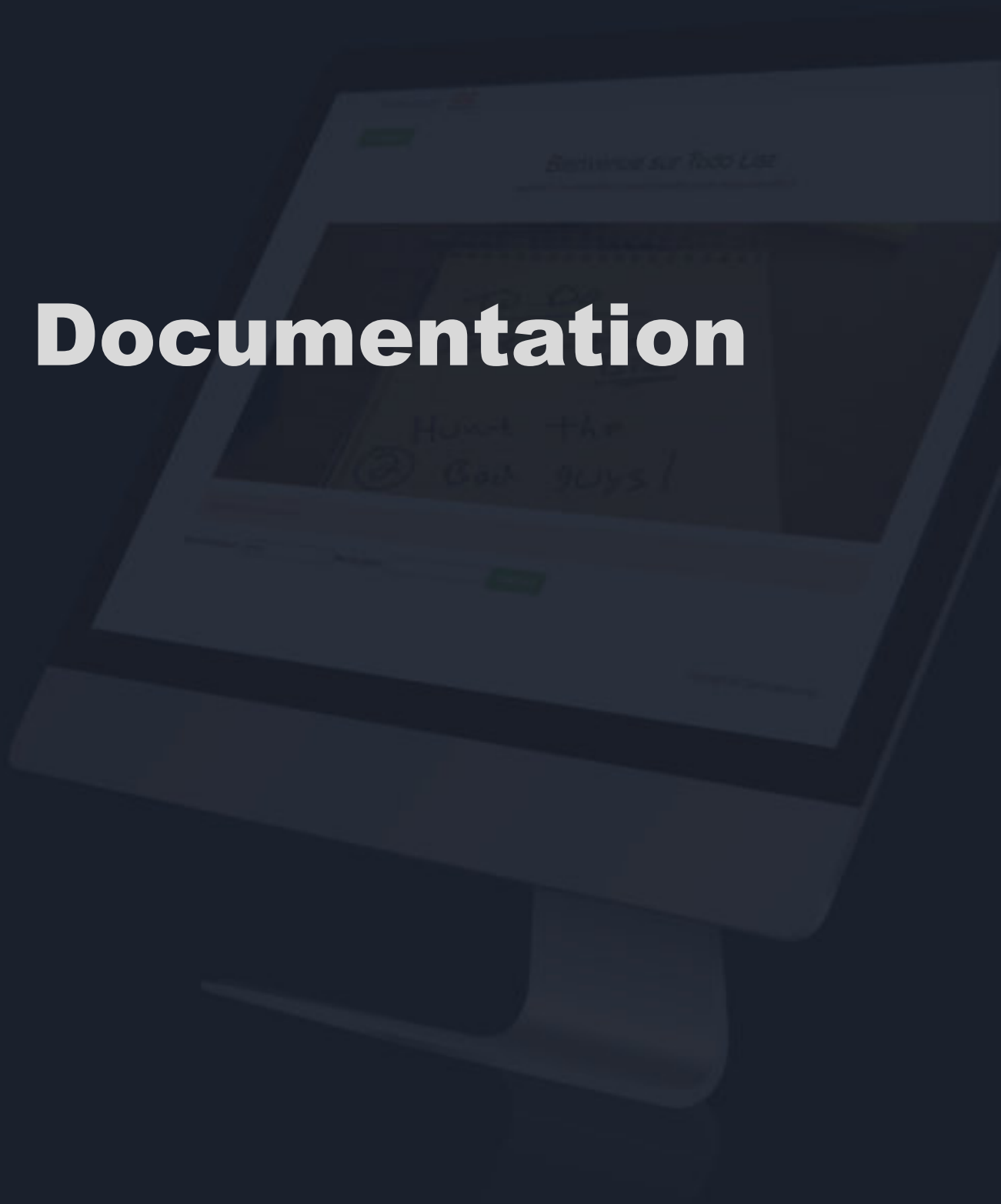


Projet 8

Améliorez une application existante de ToDo & Co

MVE ESSONE Edgard Michel

Documentation



Documentation : comment fonctionne l'authentification ?

Cette documentation vous explique comment l'implémentation de l'authentification a été faite sur l'application. Elle va vous montrer quels fichiers il faut modifier et pourquoi. Elle vous expliquera aussi comment s'opère l'authentification au travers du composant Security de Symfony.

Construction de l'entité User

La construction de l'entité User est la première étape à prendre en compte dans la création d'une authentification. L'entité User de Symfony définit la structure de notre utilisateur (username, password, role etc.). C'est à partir de cette entité que l'on construira le controller ainsi que le formulaire de connexion.

On crée une entité en passant par le terminal et la console Symfony :

~ php bin/console make:entity

Néanmoins, l'entity User n'est pas une entité comme une autre, elle doit implémenter une interface : la UserInterface qui permet d'inclure certaines méthodes que le composant security de Symfony utilise pour mettre en place le pare feu et le système d'authentification. Voici les méthodes à mettre en place obligatoirement :

```
class User implements UserInterface
{
        // ...
}
interface UserInterface
{
    /**
     * Returns the roles granted to the user.
     *
     *     public function getRoles()
     *     {
     *         return ['ROLE_USER'];
     *     }
     *
     * Alternatively, the roles might be stored on a ``roles`` property,
     * and populated in any number of different ways when the user object
     * is created.
```

```
     *
     * @return (Role|string)[] The user roles
     */
    public function getRoles();

    /**
     * Returns the password used to authenticate the user.
     *
     * This should be the encoded password. On authentication, a plain-text
     * password will be salted, encoded, and then compared to this value.
     *
     * @return string The password
     */
    public function getPassword();

    /**
     * Returns the salt that was originally used to encode the password.
     *
     * This can return null if the password was not encoded using a salt.
     *
     * @return string|null The salt
     */
    public function getSalt();

    /**
     * Returns the username used to authenticate the user.
     *
     * @return string The username
     */
    public function getUsername();

    /**
```

```
     * Removes sensitive data from the user.
     *
     * This is important if, at any given point, sensitive information like
     * the plain-text password is stored on this object.
     */
    public function eraseCredentials();
}
```

Configurer le composant Security de Symfony (security.yaml)

Le fichier security.yaml est le fichier de configuration du composant security de Symfony. Nous allons détailler les différentes parties de sa configuration.

Security.yaml - Le provider

Le provideur de Symfony permet de créer et d'authentifier des utilisateurs en toute sécurité. Il permet d'indiquer à Symfony l'objet User utilisé pour l'authentification. Il permet aussi de mettre en place un système de session au niveau de l'authentification (par ex : recharger le nom de l'utilisateur en cas de mauvais mot de passe).

Il est nécessaire dans la configuration d'indiquer au provider où aller chercher les informations utilisateurs. Dans notre cas, ce sera dans l'entité User créée juste avant :

Security.yaml - Le provider

Le provideur de Symfony permet de créer et d'authentifier des utilisateurs en toute sécurité. Il permet d'indiquer à Symfony l'objet User utilisé pour l'authentification. Il permet aussi de mettre en place un système de session au niveau de l'authentification (par ex : recharger le nom de l'utilisateur en cas de mauvais mot de passe).

Il est nécessaire dans la configuration d'indiquer au provider où aller chercher les informations utilisateurs. Dans notre cas, ce sera dans l'entité User créée juste avant :

```
providers:
        doctrine:
                entity:
                        class: App:User
                        property: username
```

La classe User doit être renseignée en suivant le namespace choisi. Dans notre cas ce sera App\Entity\User. Dans l'exemple précédent, nous utilisons un alias au niveau de doctrine pour ne pas avoir à rentrer tous le namespace :

Dans notre application nous avons choisi d'utiliser une entity Symfony pour implémenter l'authentification. Cela nous permet d'utiliser la structure de notre entity User et d'avoir une liaison directe avec la base de données. De cette manière, nous irons chercher les utilisateurs directement en base de données avec l'aide du composant Entity.

Il est aussi possible de choisir entre plusieurs options de configuration comme par exemple

users_in_memory qui permet de rentrer les informations utilisateurs directement dans le fichier security.yaml.

Pour plus d'information concernant cette partie de la configuration, vous pouvez vous rendre ici.

Security.yaml - L'encoder

La partie encoder de la configuration du composant Security permet de configurer l'encodage des mots de passe de l'application.

```
encoders:
    App\Entity\User: bcrypt
```

Nous devons ici choisir quelle entité est concernée par l'authentification et la nommer en suivant le namespace de l'entité. Dans notre cas ce sera App\Entity\User. L'option bcrypt concerne le type d'encodage choisi pour crypter les mots de passe.

Il est aussi possible de paramétrer une deuxième option cost qui permet de choisir l'importance de l'encodage effectué par le composant Security de Symfony. Mais attention à ne pas abuser de cet option car elle ralentit forcément l'application lors d'un cout d'encodage élevé.

Security.yaml - Le firewall

Le partie firewall (littéralement le pare feu) est la partie la plus importante de la configuration du security.yaml. Elle consiste à définir les règles d'authentification utilisées sur l'application. La configuration du firewall permet aussi de restreindre les accès de l'application aux utilisateurs authentifiés mais aussi de définir la marche à suivre en terme de connexion et de déconnexion.

La première partie de la configuration du firewall permet de définir les sources de l'application ignorés par le pare feu de Symfony. Par exemple, nous ne voulons pas appliquer de vérification sur les assets de l'application. Par conséquent, nous allons écrire le code suivant :

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

Ce code permet, via une regex et la clé security à false, d'ignorer les sources de l'application par le pare-feu de Symfony. Il débloque aussi l'accès à tous les outils de développement utilisés par Symfony. Une fois en production (APP_ENV passé à prod), cette partie du firewall n'est plus prise en compte.

La partie main du firewall définit le comportement du pare-feu pour tous le site.

```
main:
    anonymous: ~
    pattern: ^/
    form_login:
        check_path: login
    logout:
        path: logout
        target: homepage
```

La clé anonymous indique que le pare-feu est accessible et doit être activé quand nous ne sommes pas connectés (c'est à dire en mode "anonyme").

Le pattern à indiquer ici est beaucoup moins restrictif puisque l'on englobe toutes les routes / urls. On verra par la suite comment ne pas prendre en compte certaines urls dans le pare-feu via la clé access_control.

La 3ème clé de configuration de la partie main est le fournisseur d'authentification. Il permet de configurer la route utilisé pour se connecter. Il existe plusieurs types de fournisseur d'authentification que vous pouvez consulter ici.

Dans notre application nous allons utiliser form_login, c'est à dire que nous allons utiliser un formulaire d'authentification pour se connecter. Pour que le système de sécurité puisse reconnaitre le formulaire et travailler avec lui, il faut lui définir, dans la clé check_path, la route associée au formulaire : login.

Au préalable, il faudra définir dans un controller la route login qui se chargera d'afficher le formulaire de connexion. De cette manière, on précise aussi au système de sécurité de Symfony que la route login est accessible même en étant non connecté.

Il y a donc deux fichiers à ajouter pour créer ce formulaire de connexion :

App\Controller\UserController : permet d'ajouter le formulaire sur la page à l'aide de la méthode createForm() qui se chargera d'afficher le formulaire sur la route associé : /login

App\Form\UserType : permet de construire le formulaire de connexion à l'aide du form builder de Symfony.

Le dernière clé de configuration de la partie main est la clé logout qui définit les règles de déconnexion lié au pare-feu Symfony.

```
logout:
        path: logout
        target: homepage
```

On définit, comme le système de connexion, une route associée à la déconnexion de l'utilisateur, via la clé path. Il faudra donc ajouter dans le UserController une route pour la déconnexion car elle n'existe pas pour l'instant.

On pourra définir aussi, si ca n'a pas été fait dans le controlleur, une clé target qui permet de rediriger l'utilisateur déconnecté vers une route spécifique. Nous avons choisi de rediriger vers la page d'accueil.

Security.yaml - Access control

Comme on le précisait un peu plus haut dans la documentation, la clé de configuration access_control permet d'autoriser l'accès à certaines pages en fonction du rôle de l'utilisateur connecté (ROLE_USER, ROLE_ADMIN...). Pour cela, il faut renseigner :

Un chemin d'accès à la page (route) : path

Le role minimum de l'utilisateur pour accéder à cette page

Pour l'exemple, voici l'access_control que nous avons mis en place pour l'application :

access_control:

- { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }

- { path: ^/, roles: ROLE_USER }

Vous voyez qu'il est aussi possible de renseigner un role anonyme IS_AUTHENTICATED_ANONYMOUSLY qui permet de donner l'accès anonymement (non connecté) à la page.

Mise en place des Voters de Symfony

L'utilisateur d'access_control est conseillé dans la plupart des cas, mais si nous voulons une gestion plus poussée des rôles dans notre application (ce qui est notre cas), nous devons utiliser le système de Voter Symfony.

Un voter permet d'autoriser un utilisateur à effectuer une action particulière (ajouter une tâche, voir la liste des utilisateurs etc...), de manière beaucoup plus "maléable" qu'avec la configuration access_control.

Pour ajouter un voter, il faut passer par le terminal et la console Symfony, et définir ensuite le nom de la classe :

~ php bin/console make:voter

Par exemple, dans notre application, nous avons mis en place un système de Voter pour l'accès à la gestion des utilisateurs que nous devons restreinte au ROLE_ADMIN. Nous avons donc crée une classe : App\Security\Voter\UserVoter.

Pour mettre en place un Voter Symfony, il faut respecter une certaine configuration :

Mettre en place des actions sous la forme de constante PHP. Par ex, dans notre cas, nous voulons créer 3 accès différents sur notre application :

Voir les utilisateurs

Créer un utilisateur

Modifier un utilisateur

Nous allons donc créer 3 constantes PHP en suivant ce format :

```
const VIEW = 'view';

const CREATE = 'create';

const UPDATE = 'update';
```

Ensuite, deux méthodes sont obligatoires dans l'utilisation d'un Voter Symfony :

supports() : Cette méthode vérifie que le "sujet" passé en paramètre est bien une instance de la classe App\Entity\User. De plus, elle vérifie que l'attribut passé en paramètre correspond bien à une constante de la classe App\Security\Voter\UserVoter (dans notre cas : VIEW, UPDATE ou CREATE)

voteOnAttribute() : en fonction de l'attribut utilisé, on décidera de retourner une classe en particulier qui vérifiera que l'utilisateur a bien les accès demandés. Par exemple, si l'attribut est VIEW, on appellera une méthode canView() (qui peut s'appeller comme vous le voulez) qui vérifie les droits pour accéder à la liste des utilisateurs.

Maintenant que nous avons notre Voter Symfony configuré, il faut l'utiliser dans notre controller App\Controller\UserController.

La première chose à faire est d'instancier l'interface AuthorizationCheckerInterface pour nous permettre d'utiliser la méthode isGranted() afin de vérifier que l'utiliseur connecté a bien le bon rôle pour effectuer l'action :

```
// $this->authorization -> instance de AuthorizationCheckerInterface


if ($this->authorization->isGranted(UserVoter::CREATE, $this->getUser())) {

        // accès à la ressource

}
```

Dans cette méthode isGranted(), nous avons donc besoin de définir la constante utilisé (VIEW, UPDATE, CREATE) issue de la classe App\Security\Voter\UserVoter ainsi que l'utilisateur actuellement connecté.

La méthode renvoit true si l'utilisateur est autorisé et false dans le cas contraire.

Contribuez sur ce projet !

Ca y est, vous êtes au point sur l'authentification sur Symfony ? Vous souhaitez collaborer sur le

projet ? Merci de suivre les règles de contribution mises en place.