# Pour instaurer un système d’authentification nous avons ajouté le **Security Bundle**.

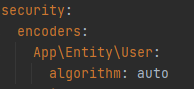
Le Security Bundle vous permet de mettre en place différent types d’authentification sur votre application. En effet, votre application peut présenter différents espaces. On peut voir ces espaces comme les pièces d’une maison. Si on imagine une colocation entre plusieurs personnes, Il y a les pièces principales qui sont ouvertes à tous. Et chaque chambre de la colocation est réservé à un colocataire. Pour avoir accès à la maison il faut posséder les clés et faire partie des colocataires. C’est une première logique d’accès. Ensuite pour entrer dans la chambre de X il faut avoir l’identité de X. C’est encore une autre logique d’accès. Pour notre application les pièces ce sont les routes. Et en fonction des pièces on met en place une logique de vérification. C’est une manière de sécuriser notre application. Dans notre application ces espaces se nomment des firewalls.

Voici différents types d’espaces que peut contenir votre application :

* /Login
* /admin

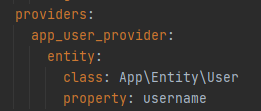
Toute la configuration du bundle se fait dans le fichier **security.yaml**. Voici sa configuration :

* **La section Encoders** : permet de définir l’encodage du mot de passe.



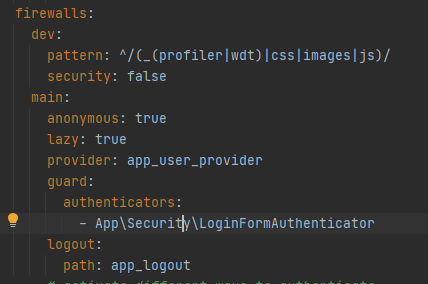
Dans la section **encoders** on indique le ***namespace*** de notre entité **User.**  L’option **algorithm** nous permet de choisir la méthode d’encodage du mot de passe. En indiquant ***auto***symfony choisie la méthode la plus puissante pour le faire.

* **La section providers** : indique au composant Security où se trouvent les données de l’utilisateur dont on souhaite vérifier les informations. On lui donne notre classe User.



**Property** nous permet d’indiquer le champ qui sera associé au mot de passe pour contrôler l’utilisateur.

* **La section firewalls :** comme expliqué plus haut ce sont les régions de votre application dont les frontières sont matérialisées par des URLs.



Il faut ici se concentrer sur la partie **main.** Au sein de cette partie on indique notre provider.Et notre classe de connexion (authenticator). L’authenticator est une classe qui nous permet de construire les méthodes de vérifications lors d’une demande de connexion. Voir la section [Comment gérer la connexion d’un utilisateur ?](#_Comment_gérer_la)

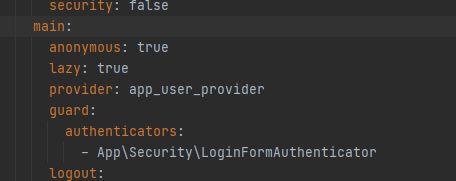
A savoir Lorsque vous utilisez un système d’authentification vous devez utiliser une entité **User**. Cette entité User sers à stocker les informations de nos utilisateurs. Vous pouvez donner n’importe quel nom à cette entité. Le principal c’est que cette entité est destinée à contenir les informations d’authentification. Et qu’elle incorpore L’**UserInterface** qui regroupe un ensemble de méthode qui vont nous permettre de gérer les informations de l’utilisateur.

## Pour Hasher les mots de passe de vos utilisateurs

Par exemple lorsque vous souhaitez Hasher le mot de passe d’un de vos utilisateurs avant l’insertion en base de données. Vous devez utiliser l’**UserPasswordEncoderInterface.** Et lui passer votre objet User et le mot de passe à hasher.

## Comment gérer la connexion d’un utilisateur ?

Pour l’application TodoList nous avons opter pour une authentification classique. C’est-à-dire la vérification d’un username et d’un mot de passe. Pour gérer la logique de vérification et les étapes de l’ensemble du système d’authentification il faut créer une classe **Authenticator**. Cette méthode va regrouper l’ensemble des méthodes qui constitue le contrôle de l’utilisateur.



**Voici les méthodes qu’elle doit comporter :**

1. **La méthode Supports :** La méthode supports permet de vérifier si la route qui est demandée nécessite le traitement d'une vérification pour l'accès à une page. Cette méthode est appelée à chaque fois qu’une requête est transmise.
2. **La méthode GetCredentials:** ressort les informations de connexion. En somme il s’agit de dire : donne-moi les informations pour la connexion. Ce que l'on va renvoyer dans la méthode getCrédentialsc'est le tableau post du formulaire d’authentification.
3. **La méthode getUser :** Doit retourner un utilisateur qui est présent dans la BDD.
4. **La méthode checkCredentials :** On vérifie que le mot de passe fourni correspond bien au mot de passe de la base de données.

**A savoir** : lorsque l'authentification échoue au sein de n'importe quelle méthode de la classe **authenticator** la classe **onAuthentificationFailure** sera appelé automatiquement en cas d'échec (avec une exception). Et dans le cas où les méthodes nécessaires à l'authentification se sont bien déroulées alors ont atterri dans la méthode **onAuthentificationSuccess**

Lorsque l'ensemble de la procédure s'est bien passé on peut par exemple renvoyer vers une page. L'utilisateur sera connecté.

**Pour obtenir les raisons de l'échec de connexion** on peut utiliser le service **AuthenticationUtils()** qui permet d'extraire les erreurs du formulaire:

## Gérer la déconnexion :

Dans cette exemple **security\_logout** est le nom de la route. Vous devez la remplacer par la route sur laquelle vous souhaitez vous déconnecter. La variable globale **app** dans twig permet d'accéder notamment à l'utilisateur. Si on souhaite savoir si l’utilisateur est connecté on interroge **app.user.id.** S’il existe un id alors il existe un utilisateur connecté. Cela nous permettra de gérer les boutons visibles à l’écran. Mais aussi les possibilités de l’utilisateur en fonction de ses droits.