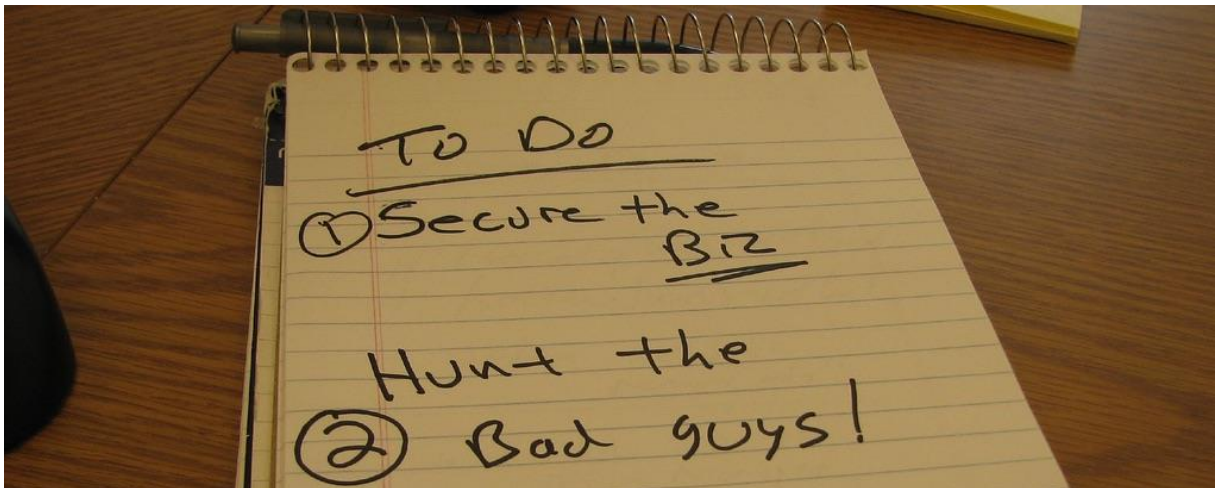


ToDo List



Authentication

Table of contents

1. Context	1
2. Symfony security system	1
2.1 Authentication	1
2.2 Authorization	1
2.3 General process	1
3. Setting up and configuring Symfony security	2
3.1 Symfony security configuration file	2
3.2 Authentication	3
3.3 Authorization	5
3.3.1 Definition of roles	5
3.3.2 Assigning roles	6
3.3.2.1 Security.authorization_checker service	6
3.3.2.2 @Security annotation in controllers	6
3.3.2.3 From a Twig view	7
3.3.2.4 Access Controls	8
3.4 Users	9
3.4.1 Encoder	9
3.4.2 Provider	9
3.4.3 Retrieve the current user	10
4. Conclusion	13

1. Context

The start-up ToDo & Co has developed an application "ToDo List" to manage its daily tasks.

This application has been developed with Symfony PHP framework.

This documentation explains how the implementation of authentication and authorization was done.

It is mainly intended for the next junior developers who will join the team, it allows among others for a beginner with the Symfony framework of:

- understand which file (s) to modify and why;
- how authentication and authorization take place;
- and where are the users stored.

2. Symfony security system

In terms of security, Symfony has separated 2 distinct mechanisms: **authentication** and **authorization**.

2.1 Authentication

Authentication is the process that will define who you are as a visitor.

Either you have not logged onto the site and you are an anonymous or you have logged in and you are a member of the site.

What handles authentication in Symfony is called a **firewall**.

2.2 Authorization

Authorization is the process that will determine if you have the right to access the requested resource (page).

It therefore acts after the firewall.

What handles authorization in Symfony is called **access control**.

For example, in our application only people with role ROLE_ADMIN can access the pages relating to the management of users, that's what access control will check.

2.3 General process

When a user tries to access a protected resource, the process is as follows:

- A user wants to access a protected resource;
- The firewall redirects the user to the login form;
- The user submits his credentials;
- The firewall authenticates the user;
- The authenticated user returns the initial request;
- The access control verifies the rights of the user and allows or not access to the protected resource.

3. Setting up and configuring Symfony security

3.1 Symfony security configuration file

The security configuration file is located in the **app / config** directory of the application and is named **security.yml**.

Here is the file of our application:

```

1  # app/config/security.yml
2
3  security:
4    encoders:
5      AppBundle\Entity\User: bcrypt
6
7    role_hierarchy:
8      ROLE_ADMIN:      ROLE_USER
9
10   providers:
11     doctrine:
12       entity:
13         class: AppBundle\User
14         property: username
15
16   firewalls:
17     dev:
18       pattern: ^/(_(profiler|wdt)|css|images|js)/
19       security: false
20
21   main:
22     anonymous: ~
23     pattern: ^/
24     form_login:
25       login_path: login
26       check_path: login_check
27       always_use_default_target_path: true
28       default_target_path: /
29     logout: ~
30
31   access_control:
32     - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
33     - { path: ^/users, roles: ROLE_ADMIN }
34     - { path: ^/, roles: ROLE_USER }
35

```

Encoders section (see 3.4 Users - 3.4.1 Encoder)

Role_hierarchy section (see 3.3 Authorization - 3.3.1 Defining roles)

Providers section (see 3.4 Users - 3.4.2 Provider)

Firewalls section (see 3.2 Authentication)

Access_control section (see 3.3 Authorization - 3.3.2 Assigning roles)

3.2 Authentication

A firewall tries to verify that you are the one you claim to be.

Firewalls from our application - **firewalls** section:

```
16     firewalls:
17         dev:
18             pattern: ^/(_(profiler|wdt)|css|images|js)/
19             security: false
20
21         main:
22             anonymous: ~
23             pattern: ^/
24             form_login:
25                 login_path: login
26                 check_path: login_check
27                 always_use_default_target_path: true
28                 default_target_path: /
29             logout: ~
```

The **dev** firewall is not important, it just makes sure that the Symfony development tools - which are under URLs like `/_profiler` and `/_wdt` are not blocked by your security.

The **main** firewall handles all other URLs:

- **anonymous**: `~` accepts anonymous users. We protect our resources through roles.
- **pattern**: `^ /` is a URL mask.
This means that all URLs starting with `"/` are protected by this firewall.
- **form_login** is the authentication method used for this firewall. It corresponds to the classical method, via an HTML form:

ToDo List

```
1 #app/Resources/views/security/login.html.twig
2
3 {% extends 'base.html.twig' %}
4
5 {% block header_img %}{% endblock %}
6
7 {% block body %}
8     <div class="col-md-12">
9         <div class="col-md-offset-4 col-md-4 login">
10             {% if error %}
11                 <div class="alert alert-danger" role="alert">{{ error.messageKey|trans(error.messageData, 'security') }}</div>
12             {% endif %}
13             <div class="text-center logo-login">
14                 
15                 <p class="title-login text-center">ToDo List App</p>
16             </div>
17
18             <form action="{{ path('login_check') }}" method="post">
19                 <div class="form-group">
20                     <input type="text" class="form-control" id="username" name="_username" placeholder="Nom d'utilisateur" value="{{ last_username }}" />
21                 </div>
22                 <div class="form-group">
23                     <input type="password" class="form-control" id="password" placeholder="Mot de passe" name="_password" />
24                 </div>
25                 <button class="form-control btn-login" type="submit">Se connecter</button>
26             </form>
27         </div>
28     </div>
29 {% endblock %}
```

Its options are:

- **login_path**: **login** is the route of the login form. This route uses the LoginAction method of the SecurityController.

```
1 <?php
2 #AppBundle\Controller\SecurityController.php
3 + /*...*/
4
11
12 namespace AppBundle\Controller;
13
14 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
15 use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
16
17 class SecurityController extends Controller
18 {
19     /**
20      * @Route("/login", name="login")
21      */
22     public function loginAction()
23     {
24         $authenticationUtils = $this->get('security.authentication_utils');
25
26         $error = $authenticationUtils->getLastAuthenticationError();
27         $lastUsername = $authenticationUtils->getLastUsername();
28
29         return $this->render( view: 'security/login.html.twig', array(
30             'last_username' => $lastUsername,
31             'error'         => $error,
32         ));
```

ToDo List

- **check_path: login_check** is the validating route of the login form, it is on this route that the credentials entered by the user on the previous form will be checked.
- **always_use_default_target_path: true**: Ignore the previously requested URL and automatically redirect to the default page: default_target_path.
- **Logout** makes disconnection possible. Indeed, by default it is impossible to disconnect once authenticated.

The process is as follows:

When the security system (here, the firewall) initiates the authentication process, it will redirect the user to the login form (the login route).

This form will have to send the values (username and password) to the route (here, login_check) that will support the form management.

It is the Symfony security system that will take care of the management of this form.

Specifically, the loginAction function of the SecurityController controller executes the login route.

We do not define a function to execute for the login_check route, Symfony will catch the visitor's request on the login_check route, and manage the authentication itself.

If successful, the visitor will be authenticated.

In case of failure, Symfony will send it back to our login form to try again.

Do not forget the definition of roads

We must remember to create the login, login_check and logout routes.

These are mandatory routes, and if they are forgotten it is quite possible to have 404 errors in the middle of the authentication process.

Firewalls do not share

In the case of using multiple firewalls, you should know that they share nothing with each other. Thus, if we are authenticated on one, we will not necessarily be on the other, and vice versa. This increases security during complex setup.

We'll put / login_check behind the firewall

It must be ensured that the URL of the check_path (here, / login_check) is well behind the firewall that we use for the login form (here, main).

Indeed, it is the road that allows authentication at the firewall. But since firewalls do not share anything, if this route does not belong to the firewall we want, we will be entitled to an error.

In our case, the pattern: ^ / of the main firewall takes the URL / login_check, so it's OK.

3.3 Authorization

3.3.1 Définition of roles

The notion of "role" is central to the authorization process.

ToDo List

One or more roles are assigned to each user, and to access the resources it is requested that the user has one or more roles.

Thus, when a user tries to access a resource, the access controller checks whether he has the role or roles required by the resource. If so, access is granted. Otherwise, access is denied.

The **role_hierarchy** section in the security.yml file of the configuration establishes the role hierarchy.

```
7  role_hierarchy:
8  ROLE_ADMIN: ROLE_USER
```

We defined 2 roles:

- ROLE_ADMIN
- ROLE_USER

Thus, the ROLE_USER role is included in the role ROLE_ADMIN. This means that if your page requires the ROLE_USER role, and a user with the ROLE_ADMIN role tries to access it, it will be allowed, because by having the ROLE_ADMIN role, it also has the ROLE_USER role.

The names of the roles do not matter, except that they must start with "ROLE_".

3.3.2 Assigning roles

Access control will take care of determining if the visitor has the right rights (roles) to access the requested resource.

There are 4 methods for assigning roles to users.

3.3.2.1 Security.authorization_checker service

From a controller or any other service, access the security.authorization_checker service and call the isGranted method.

The other 3 methods go through this service.

In our application we did not use this method.

3.3.2.2 @Security annotation in controllers

The @Security annotation comes from the SensioFrameworkExtraBundle bundle.

The annotation method is used to secure one or more controller methods. **This is the method recommended by Symfony best practices.**

We used this method in our application in the UserController to specify that all the methods of this controller were limited to ROLE_ADMIN:


```

1  <?php
2
3  + /* ... */
11
12  namespace AppBundle\Controller;
13
14  use AppBundle\Entity\User;
15  use AppBundle\Form\Type\UserType;
16  use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
17  use Symfony\Bundle\FrameworkBundle\Controller\Controller;
18  use Symfony\Component\HttpFoundation\Request;
19  use Sensio\Bundle\FrameworkExtraBundle\Configuration\Security;
20
21  /**
22   * @Security("has_role('ROLE_ADMIN')")
23   */
24
25  class UserController extends Controller

```

3.3.2.3 From a Twig view

This method is very convenient for displaying different content depending on the roles of the users.

The Twig method is used to secure the display.

For this, Twig has an **is_granted ()** function that is actually a shortcut to run the `servicesecurity.authorization_checker's isGranted ()` method.

We used this method in the Twig views:

- **App/Resources/base.html.twig:**

```

60  {% if is_granted('ROLE_ADMIN') and 'user_list' != app.request.attributes.get('_route') and 'login' != app.request.attributes.get('_route') %}
61  <li class="dropdown"><a href="" class="dropdown-toggle btn-nav" data-toggle="dropdown" id="dropdownTasks" aria-haspopup="true" aria-expanded="true">Gestion
62  <ul class="dropdown-menu" aria-labelledby="dropdownTasks">
63  <li><a href="{{ path('user_create') }}" class="btn-nav">Créer un utilisateur</a></li>
64  <li><a href="{{ path('user_list') }}" class="btn-nav">Liste des utilisateurs</a></li>
65  </ul>
66  </li>
67  {% endif %}

```

Which makes it possible to have in the navigation menu access to the management of the users only for the users having the `ROLE_ADMIN`.

ToDo List

```
91     {% if is_granted("IS_AUTHENTICATED_REMEMBERED") %}
92         <span class="connected">{{ app.user.username }}</span>
93         <a href="{{ path('logout') }}"><i class="fa fa-sign-out-alt" aria-hidden="true"></i></a>
94     {% else %}
95         <a href="{{ path('login') }}"><i class="fa fa-sign-in-alt"></i></a>
96     {% endif %}
```

This will display the username for authenticated users.

- App/Resources/views/default/index.html.twig:

```
15     {% if is_granted('ROLE_ADMIN') %}
16         <div class="row btn-index-2">
17             <div class="col-md-offset-2 col-md-4">
18                 <a href="{{ path('user_create') }}" class="btn btn-global">Créer un nouvel utilisateur</a>
19             </div>
20             <div class="col-md-4">
21                 <a href="{{ path('user_list') }}" class="btn btn-global">Consulter la liste des utilisateurs</a>
22             </div>
23         </div>
24     {% endif %}
```

This makes it possible to display the buttons "Créer un nouvel utilisateur" and "Consulter la liste des utilisateurs" only for the users having the role ROLE_ADMIN.

3.3.2.4 Access Controls

The access control method is used to secure URLs. It is configured in the **security.yml** security configuration file, in the **access_control** section.

We used this method and determined the following accesses - **Access_control** section:

```
31     access_control:
32         - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
33         - { path: ^/users, roles: ROLE_ADMIN }
34         - { path: ^/, roles: ROLE_USER }
```

So all the URLs starting with:

- **/ login** requires IS_AUTHENTICATED_ANONYMOUSLY role, so anonymous people can access it
- **/ users** requires ROLE_ADMIN
- **/** requires ROLE_USER

Do not secure the login form

If the form is secure, newcomers will not be able to authenticate.

In this case, we must be careful that the page **/ login** does not require any role.

3.4 Users

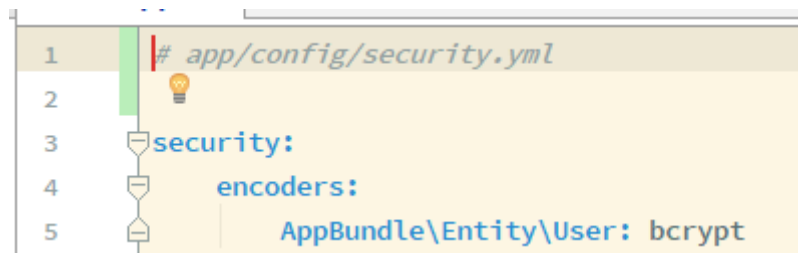
Users are stored in database.

In our application, a user is represented by our **User** entity (AppBundle / Entity / User) that implements the `UserInterface` interface.

3.4.1 Encoder

For our application, we chose to encode the users' passwords with the '**bcrypt**' method, the 'User' entity representing the users.

Encoders section:



```
1 # app/config/security.yml
2
3 security:
4     encoders:
5         AppBundle\Entity\User: bcrypt
```

3.4.2 Provider

It is a “user provider”.

Firewalls ask the providers to retrieve users and identify them.

In our application, users are stored in the database and the application retrieves them and identifies them using the 'username' property of the 'User' entity.

A “user provider” is a class that implements the `UserProviderInterface` interface, which contains just three methods:

- **loadUserByUsername(\$username)**, that loads a user from a username;
- **refreshUser(\$user)**, that refreshes a user with the original values;
- **supportsClass()**, that determines which class of users manages the provider.

Symfony already has three types of providers:

- **Memory**: uses the users defined in the configuration;
- **Entity**: simply use an entity to provide users, **this is the one we used**;
- **Id**: allows you to use any service as a provider, specifying the name of the service.

So we defined the provider for our User entity as well - **Providers** section:

ToDo List

```
10 providers:
11     doctrine:
12         entity:
13             class: AppBundle\User
14             property: username
```

- doctrine: provider name
- entity: provider type
- class: class of the entity to use
- property: attribute of the class that serves as identifier

It would be necessary to add our provider in the firewall to indicate to it that it is necessary to use this one, but if we do not do it the first provider will be automatically used.

```
16 firewalls:
17     dev:
18         pattern: ^/(_(profiler|wdt)|css|images|js)/
19         security: false
20
21     main:
22         anonymous: ~
23         provider: doctrine
24         pattern: ^/
25         form_login:
26             login_path: login
27             check_path: login_check
28             always_use_default_target_path: true
29             default_target_path: /
30         logout:
```

In our application we did not add it in the firewall.

The security layer is now fully operational and uses users stored in the database.

3.4.3 Retrieve the current user

To retrieve information about the current user, whether anonymous or not, use the service **security.token_storage**.

This service has a **getToken ()** method, which allows you to retrieve the current security session (not to be confused with the classic session, available via `$request->getSession ()`).

This token is worth **null** if we are out of a firewall.

And if we are behind a firewall, then we can recover the current user with **\$token->getUser ()**.

ToDo List

From the controller

The controller has a shortcut to retrieve the current user, this is the `$this->getUser()` method. This method returns:

- **Null** if the request is not behind a firewall, or if the current user is anonymous;
- **An instance of User** the rest of the time (authenticated user behind a firewall and non-anonymous).

This method is used in TaskController to retrieve the current user:

- `createAction` function (line 52):

```
43  /**
44   * @Route("/tasks/create", name="task_create")
45   * @param Request $request
46   * @return \Symfony\Component\HttpFoundation\RedirectResponse|\Symfony\Component\HttpFoundation\Response
47   */
48  public function createAction(Request $request)
49  {
50      $task = new Task();
51
52      $user = $this->getUser();
53      $task->setUser($user);
54
55      $form = $this->createForm( type: TaskType::class, $task);
56
57      $form->handleRequest($request);
58
59      if ($form->isValid()) {
60          $entityManager = $this->getDoctrine()->getManager();
61
62          $entityManager->persist($task);
63          $entityManager->flush();
64
65          $this->addFlash( type: 'success', message: 'La tâche a été bien été ajoutée.');
```

- `deleteTaskAction` function (line 122):

ToDo List

```
114  /**
115      * @Route("/tasks/{id}/delete", name="task_delete")
116      * @param Task $task
117      * @return \Symfony\Component\HttpFoundation\RedirectResponse
118      * @throws \Exception
119      */
120  public function deleteTaskAction(Task $task)
121  {
122      $user = $this->getUser();
123
124      if (!$task->canBeDeletedBy($user)) {
125          throw new Exception( message: "Vous n'avez pas la permission de supprimer cette tâche.");
126      }
127
128      $entityManager = $this->getDoctrine()->getManager();
129      $entityManager->remove($task);
130      $entityManager->flush();
131
132      $this->addFlash( type: 'success', message: 'La tâche a bien été supprimée.');
```

```
133
134      return $this->redirectToRoute( route: 'task_list');
```

```
135  }
```

```
136 }
```

From a twig view

We have easier access to the user directly from Twig via `{{app.user}}` used in the views:

- task/list.html.twig (line 38):

```
37  <div class="caption col-sm-1">
38      {% if task.canBeDeletedBy(app.user) %}
39      <form action="{{ path('task_delete', {'id' : task.id }) }}">
40          <button class="btn btn-delete btn-sm" title="Supprimer la tâche"><i class="fas fa-times-circle"></i></button>
41      </form>
42      {% endif %}
43  </div>
```

- base.html.twig (lines 68, 71 et 92):

```
68      {% if not app.user and 'login' != app.request.attributes.get('_route') %}
69      <li><a href="{{ path('login') }}" class="btn-log">Connexion</a></li>
70      {% endif %}
71      {% if app.user and 'logout' != app.request.attributes.get('_route') %}
72      <li><a href="{{ path('logout') }}" class="btn-log">Se déconnecter</a></li>
73      {% endif %}
74  </ul>
75  </div><!--/.nav-collapse -->
76  </div>
77  </nav>
78  <nav class="navbar navbar-default navbar-fixed-bottom">
79      <div class="container-fluid navbar-responsive">
80          <div class="row">
81              {% if 'login' != app.request.attributes.get('_route') %}
82              <div class="col-xs-4">
83                  {% if 'homepage' != app.request.attributes.get('_route') %}
84                  <a href="{{ path('homepage') }}"><i class="fa fa-home" aria-hidden="true"></i></a>
85                  {% endif %}
86              </div>
87              <div class="col-xs-4">
88                  <a href="{{ path('task_list') }}"#list_tricks"><i class="fa fa-bookmark" aria-hidden="true"></i></a>
89              </div>
90              <div class="col-xs-4">
91                  {% if is_granted("IS_AUTHENTICATED_REMEMBERED") %}
92                  <span class="connected">{{ app.user.username }}</span>
```

ToDo List

As the controller this returns either null or an instance of User.

4 Conclusion

Security under Symfony is very advanced, we can control it very finely, but especially very easily.

Do not hesitate to refer to Symfony's official documentation:

<https://symfony.com/doc/3.4/security.html>