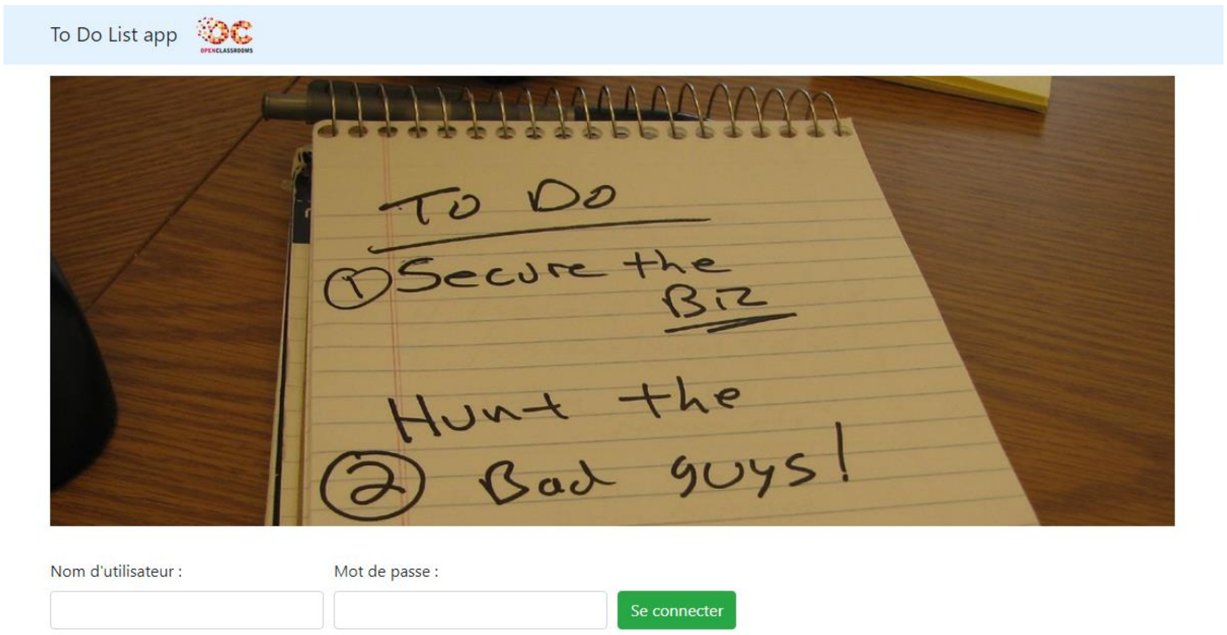
To Do List Application - **Symfony** Framework, version 4.4.

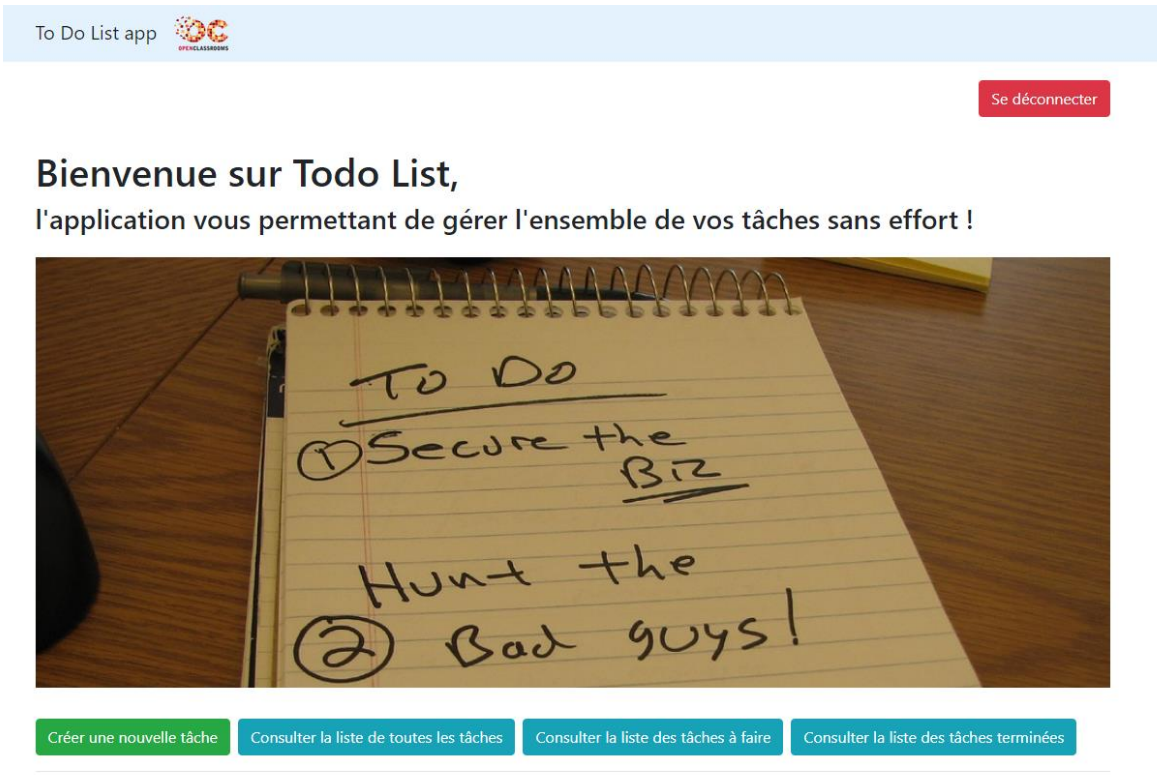
Authentication Implementation

➢ User Point of view

An unanthentified user is automatically redirected on the login page.



The user authenticates using the form login. After providing his user name and his password, he click on the « Se connecter » button. If the credentials are valid, the user is now logged and he is redirected to the task management page:


To Do List app
Se déconnecter
To Do
① Secure the Biz
Hunt the
② Bad guys!
Créer une nouvelle tâche
Consulter la liste de toutes les tâches
Consulter la liste des tâches à faire
Consulter la liste des tâches terminées

Bienvenue sur Todo List,

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

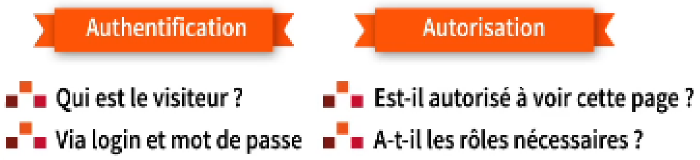
➢ Preambule

The application protection requires the protection of URLs, and then of routes associated to these URLs. The route are defined by methods in controllers (the classes defined in the src/Controller directory):

- A part of these routes can be accessible to the general public [unauthenticated visitors]
- An other part of these routes can be accessible only to some authenticated visitors.

So in matters of security, we talk about two notions:

1- **The authentication**: which identifies the visitor.
2- **The authorization:** which allows to know if the authenticated visitor has the right to access to a feature or to a page.

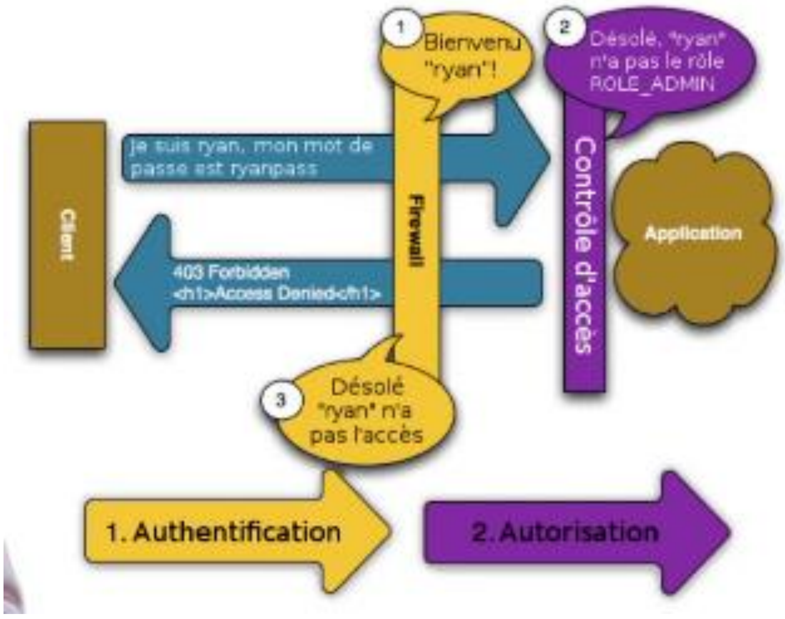


Authentification
- Qui est le visiteur ?
- Via login et mot de passe

Autorisation
- Est-il autorisé à voir cette page ?
- A-t-il les rôles nécessaires ?

These two consecutive notions: **first** the visitor is authenticated, **then** the authorization determines the access rights to the asked URL.

For example, here below a schema which explain that an authenticated user doesn't have rights to access to a page which requires the « ROLE_ADMIN » role.


1 Bienvenu "ryan"!
2 Désolé, "ryan" n'a pas le rôle ROLE_ADMIN
Client
Je suis ryan, mon mot de passe est ryanpass
Firewall
Contrôle d'accès
Application
403 Forbidden <h1>Access Denied</h1>
3 Désolé "ryan" n'a pas l'accès
1. Authentification
2. Autorisation

Images form the OpenClassrooms course « Developping your site with the Symfony 3 framework» by Fabien Potencier, creator of Symfony Framework

In the end, here are steps for a user who try to access to a protected resource:

1. A visitor want to access to a protected resource
2. He's redirect to the login form by the **firewall**.
3. The user submit his credentials (e.g. login and password)
4. The firewall authenticates the user by confirming his credentials.
5. The initial request is returned by the authenticated user.
6. The access-control verifies the user rights, and authorizes or not the access to the protected resource.

The security process always follows these steps but the mode of authentication can vary. Indeed, depending of your needs, there are many ways to authenticate visitors (banal login form, authentication with Google, etc). You must remember that the chosen method doesn't affect your code in controllers.

In this document, we will explain the implementation of the application's authentication, which concerns points 1 to 4 of the steps listed above.

NB : To know more on authorization in Symfony, you can consult the Symfony's official documentation.

➢ Authentication in Symfony

The authentication is the way to know who your visitor is.

There is two possibilities:

→ Either the visitor is **anonymous** because he is not identified,

→ Either the visitor is **member** because he is identified (via a form login or via a cookie)

The authentication procedure will therefore determinate if the visitor is anonymous or member of the site.

It's the **firewall** which manage the authentication in Symfony.

So, access to certain parts of the site can only be restricted to visitors who are members. In other words, the visitor will have to be authenticated so that the firewall allows him to pass.

For example, in To Do List application, only authenticated visitors (members) have access to task management pages (page that lists tasks, page to modify a task, etc).

NB: After the authentication, comes the authorization which manage acces rights of different member visitors. In Symfony, the authorization is managed by the **access-control**.

For example, in To Do List application, it's the authorization which manage the fact that only a visitor with the ROLE_ADMIN role can access to user's management pages.

➢ Authentication Implementation

1- Installation of the Symfony security feature

To manage security in Symfony, we will first ensure the presence of the Security component: **symfony/security-bundle.**

In application using Symfony Flex, we execute the following command to install the Security component before using it:

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

2- Creation of the User class :

No matter *how* you will authenticate (e.g. login form or API tokens) or *where* your user data will be stored (database, single sign-on), the next step is always the same: create a “User” class.

The *only* rule about your User class is that it *must* implement the Symfony\Component\Security\Core\User\UserInterface interface of the **Security** component.

In To Do List application, users are **stored** in **database**. So here, the User class is an entity which allows to store user data in database.

Here is the link to the **User** class entity contained in the **src/Entity/** directory; https://github.com/CarolineDirat/P8ImproveToDo/blob/master/src/Entity/User.php

Notes that we use Doctrine ORM (**Doctrine\ORM\Mapping** as **ORM**) to persist users;

→ The User class if notified as entity with the **@ORM\Entity** and **@ORM\Table** annotations to match the “user” table in database;

→ And the User class properties are notified by the **@ORM\Column** annotation to match to each one a field in the “user” table, in database.

NB: The **@Assert** annotation (**Symfony\Component\Validator\Constraints** alias**)** allows to define the validation constraints checked when creating a user, and then doesn’t concern the authentication.

NB: We can easily create an user thanks to MakerBundle, with the command:

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

We can then specify the User class name, if the User will be stored in database, the property corresponding to the login and if this app need to hash/check user passwords:

```
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
```

If User class is an entity (as in our case), you can use the make:entity command to add properties:

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

Also be sure to make and execute a migration for the new entity:

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

3- Authentication Configuration

A configuration file is dedicated to security, it’s the **security.yaml** file which is in the **config/packages** directory of the Symfony application.

Here is the security.yaml of our 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 }
```

The implementation of authentication now requires the definition of some parts of this configuration file: **security.providers**, **security.encoders** and **security.firewalls.**

a- The « User Provider » : 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'
    # ...
```

To identify users, the firewall ask a user **provider**.

In our case, users are stored in database, via the User entity. The "**users**" provider is then specified to allow us to get users from the database. The "**users**" provider must know:

→ The User entity class which represent users:

providers.users.entity.class**: 'App\Entity\User'**

→ The property of the User class which is used as credential

providers.users.entity.property**: 'username"**

b- Password hashing : 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'
# ...
```

Here we configure how hash the password, that cannot be stored as is in the database for obvious security reasons.

Now that Symfony knows *how* hash the password, you can use the **UserPasswordEncoderInterface** service to hash the password before storing users in the database.

NB: In To Do List, it's an **entity listener** « UserPasswordListener » which is in charge to hash the password via the UserPasswordEncoderInterface service, just before the first persist of the user in the database (when creating user so). For this, the entity listener listens to the prePersit Doctrine event, as it's defined in its statement in the config/service.yaml file.

c- Authentication et Firewall (pare-feu) – explanations `security.firewall`

A **firewall** is your authentication system: it defines *how* your users will be able to authenticate (for exemple, form login, API token, etc).

Only one firewall is enabled with each request: Symfony use the **pattern** key to find the first match (you can also match by host or other things).

```yaml
# 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
               # ...
```

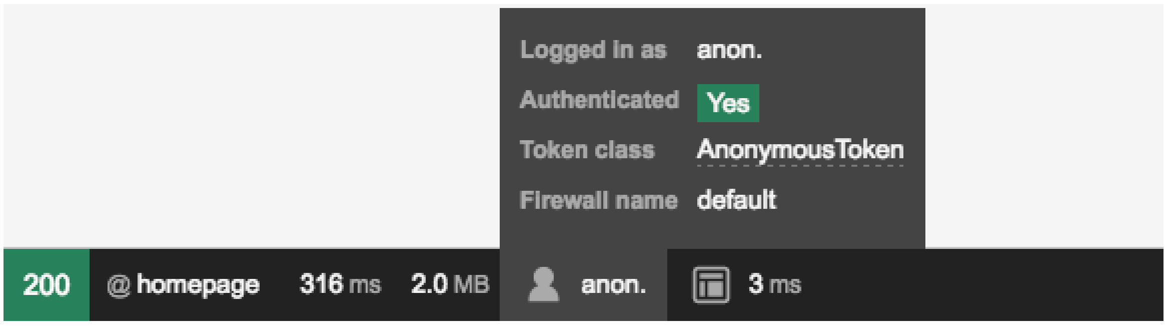
The "**dev**" firewall is rather a fake firewall: it makes sure that you don't accidentally block Symfony's dev tools - which live under URLs like /_profiler and /_wdt.

The "**main**" firewall configure the authentication method of the application. All *real* URLs are handled by the main firewall:

- **no pattern key** means it matches ***all* URLs**
- We specify that the **provider** to use is "**users**" (with `provider: users`)

A firewall can have many modes of authentication, in other words many ways to ask the question "Who are you?". Often, the user is unknown (i.e. not logged in) when they first visit your website. The anonymous mode, if enabled, is used for these requests.

In fact, if you go to the homepage right now, you *will* have access and you'll see that you're "authenticated" as **anon.**. The firewall verified that it does not know your identity, and so, you are anonymous:



It means any request can have an anonymous token to access some resource, while some actions (i.e. some pages or buttons) can still require specific privileges. A user can then access a form login without being authenticated as a unique user (otherwise an infinite redirection loop would happen asking the user to authenticate while trying to doing so).

Then it's the authorization system that will allow to refuse access to some URL, controllers or part of templates.

NB: The **lazy anonymous** mode prevents the session from being started if there is no need for authorization (i.e. explicit check for a user privilege). This is important to keep requests cacheable (see HTTP Cache).

NB: if you do not see the toolbar, install the profiler with:

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

Now that we understand the firewall, the next step is to create a way to authenticate !

d- The authentication method configured in the firewall

Authentication in Symfony can feel a bit "magic" at first. That's because, instead of building a route & controller to handle login, you'll activate an ***authentication provider***: some code that runs automatically *before* your controller is called.

Symfony has several built-in authentication providers. The To Do List application use the **form login** authentication provider, that handles a form login POST automatically. It is defined in the "main" firewall:

```
# 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

Now, when the security system initiates the authentication process, it will redirect the user to the login form on the "/login" URL, which will be handled in **SecurityController** with the "login" route (defined in the form_login configuration in security.yaml):

```php
<?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,
            ]
        );
    }
}
```

Don't let this controller confuse you. As you'll see in a moment, when the user submits the form, the security system automatically handles the form submission for you. If the user submits an invalid username or password, this controller reads the form submission error from the security system, so that it can be displayed back to the user.

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

In other words, your job is to *display* the login form and any login errors that may have occurred:

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

But the security system itself takes care of checking the submitted username and password and authenticating the user.

Finally, here is the **login form** of To Do List:

```
{# /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 %}
```

The form can look like anything, but it usually follows some conventions:

→ The **<form>** element sends a POST request to the login route, since that's what you configured under the form_login key in security.yaml;

→ The username field has the name **_username** and the password field has the name **_password**.

Notice at the red arrow that a hide field allows to generate a CSRF token thanks to the Twig **csrf_token()** function.

Also, we configure the **CSRF token provider** used in the login form in our security configuration:

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

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

By default, the HTML field must be called "**_csrf_token**" and the string used to generate the value must be "**authenticate**":

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

Finally, we define two other options to the **form_login** authentication provider:

```
# 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
             # ...
```

The **default_target_path** option allows to define the page to which the user is redirect if no previous page has been stored in the session. Here it's the home page with the "/" URL.

The **always_use_default_target_path** boolean option allows to ignore the URL previously asked and always redirect to the default page (Here the home page with the "/" URL).

And There you go !

When you submit the login form, the security system verify automatically the credentials of the user et authenticate the user, or in case of failure redirect the user to the form login with displaying of the identified error (invalid CSRF token or invalid credentials).

To review the all process :

1- The user try to access to a protected resource
2- The firewall run the authentication process redirecting the user to the form login (/login)
3- The /login page display the login form via the "login" route defined in the SecurityController
4- The user submit the login form of the /login page.
5- The security system intercept the request, verify submitted credentials, authenticate the user if they are correct and return the user to the login form if they are not.

Once the user is authenticated, he will no longer go through the login form to access to URLs protected by the firewall, but the access-control will verify his rights on each request.

4- Logging out

To enable logging out, activate the logout config parameter under your firewall: you defines the route of logging out (**logout.path:** logout):

```yaml
#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
```

Then, we create a route for this URL (but not a controller).

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

And that's all !

By sending a user to the **app_logout** route (i.e. to /logout) Symfony will un-authenticate the current user and redirect them to the /login URL (thanks to the option: **logout.target: login,** in the security.yaml file).

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: *Need more control of what happens after logout? Add a* ***success_handler*** *key under* ***logout*** *and point it to a service id of a class that implements Symfony\Component\Security\Http\Logout\**LogoutSuccessHandlerInterface**.*

5- Access to the User object of the authenticated user:

Regardless of the authentication method, the User object of the authenticated user is accessed in the same way:

- **From a controller** with the **getUser()** method

```php
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());
}
```

- **From a service** with the **Symfony\Component\Security\Core\Security** service

```php
// 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();
    }
}
```

- From a template with the **app.user** variable thanks to the the Twig global app variable:

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

➢ Conclusion

Finally, the files concerned by the authentication are:

- **config/packages/security.yaml :** the file that configure the application security, with three elements :
  - → **security.encoders** that define the method to encode the password.
  - → **security.providers** that define the users provider.
  - → **security.firewalls** that define the authentication method, and define the authentication provider and URLs managed by the firewall.
- **scr/Controller/SecurityController.php :** the file which implement the login route defined in the firewall
- the login form in the **templates/security/login.html.twig** template
- and eventually, **config/routes.yaml** to define the logging out route defined in the firewall.

The authentication method configured in security.firewalls does not affect code in controllers. That means that you can change it without impacting the code running in the controllers.

*For example, to have complete control over your login form, Symfony recommends creating a Forms Login Authentication with Guard. As explained in the official Symfony documentation, this will involve creating a **Guard authenticator** named "LoginFormAuthenticator", inheriting from AbstractFormLoginAuthenticator and implementing PasswordAuthenticatedInterface. And in security.yaml, you will have to modify **security.firewalls.main** to define the **guard** authentication provider instead of **form_login**.*

Once the authentication method is changed, you will not have to change the code of the controllers, nor the way to access the authenticated user (in a controller, service or model).

To go further, you have the official Symfony documentation