Authentication

Registration process

Creating the user

In order to register, the user must click on the button. This starts the register function, located in src\Controller\RegistrationController.php. The user class is instanciated, and a relevant form is created. This form is generated using the RegistrationFormType class. We the enter the if ($form->isSubmitted() && $form->isValid()) condition. This structure is very common in forms for symfony. It is used to check if the form has been submitted and if it is valid. If it is, we enter the if condition. The user is then saved in the database using the $entityManager->flush(); function. If the form is not submitted and valid, that means we are simply loading the page for the first time, and we don't need to do anything. We simply return the form.
Once we enter the if, we use the UserPasswordHasherInterface to hash the password. This is a security measure that ensures that the password will not be stored in plain text in the database. The user is then persisted, this signals that the user is ready to be saved in the database. The flush function is then called, which actually saves the user in the database. Note that if the initial roles are set in this controller, the rest of the information (photo and full name) are set in the User entity by using the information provided in the form.

Email confirmation

Then an email is sent to the email provided to the form. This email will contain a link that will allow the user to confirm his email. This is done using the EmailVerifierInterface. This interface is injected in the constructor of the RegistrationController. The email is sent using the $emailVerifier->sendEmailConfirmation() function. After the email is sent, a flash message is generated and the user is redirected to the home page and can login. The next step will be to get verified by clicking on the link in the email.

Email confirmation link

Once the user clicks on the link, he is redirected to the verifyUserEmail. The suer has to be connected with the right email for this to work. The verifyemailhelper then uses the validateEmailConfirmation function to handle the token provided by the link, and the user is set to verified and saved into the database. The user is then redirected to the home page and can login.

Login process

This happens in LoginController.php. You must use the template in security/login.html.twig. The form is already created and the controller is already set up. Note that src\Controller\LoginController.php is only there to provide the template. The real logic of the login comes form vendor\symfony\security-http\Authentication\AuthenticationUtils.php This dependency is injected in the index function of LoginController.php.

User storage

When you register, you use the src\Controller\RegistrationController.php and create a new src\Entity\User.php. This entity has its method used to store the information, for instance:

```
$user->setPassword(
                $userPasswordHasher->hashPassword(
                $user,
                $form->get('plainPassword')->getData()
                )
        );
```

This is used to hash the password using the UserPasswordHasherInterface, which means that instead that having a plain text password in the database, you will have a hashed password. This is a an important security measure because if for some reason your database gets hacked, the hacker will not be able to see the actual password of the users.

The user is saved when the $entityManager->flush(); is called. If you want to see the user you created, you need to go to the database (using heidiSQL for instance) and look for the user table. You will see the user with all the information you provided.

It should look like this:

```
user
---
| | | | | | | |
| ---: | --- | --- | --- | ---: | --- | --- |
| 67 | anonymous@gmail.com | ["ROLE_USER"] |
$2y$13$slPqCRSJzU7ITw2K.9USI.zes298ofakeencodedpasswordmsQXO | 1 | John Doe |
https://static.wikia.nocookie.net/shadowsdietwice/images/d/d1/Withered_Red_Gourd.png |
```

Security.yaml settings

It is crucial to understand this configuration file, because it handles many security aspects of the application. It is located in config/packages/security.yaml. It is used to configure the security system of the application.

password_hashers: this commands the selection of the password hashers for the application. The auto parameters makes it choose the best one for the application. If you wanted, you could select a specific one, such as bcrypt or argon2i.

providers: a provider is a way to get the user information. In this case, we are using the entity provider, which means that the user information is stored in the database

```
    providers:
    app_user_provider:
    entity:
        class: App\Entity\User
        property: email
```

The entity provider is configured in the entity section. The entity provider is configured to use the User entity, and the property that will be used to identify the user is the email. This means that when you login, you will need to provide the email (and the password). The password will be hashed and compared to the hashed password in the database. If they match, you will be logged in. If you wanted to use a different property to identify the user, you could use the username property instead. You would then need to change the login form to use the username instead of the email. You would also need to change the User entity to make sure that the username is unique.

firewalls: You may already know that a firewall is a software that is used to block potentially problematic connections. You have one for your computer. the pattern

pattern: ^/(_(profiler|wdt)|css|images|js)/

means that when we are in this pattern of url (all of the js, css and images for the _profiler), we disable the security. This is used to allow the profiler, the symfony bar, to be displayed. If we didn't do that, we would not be able to see the profiler bar or the links to the profiler in the request list for instance.

main: the main section is effective in all cases. the lazy true options means that we load the user when you fetch it, not just from the session. The provider app_user_provider is what we use to access the user.

```
form_login:
        # "app_login" is the name of the route created previously
        login_path: app_login
        check_path: app_login
    logout:
        path: app_logout
```

This is used to configure where the user will be redirected when he logs in and when he logs out. The login_path is where the user will be redirected when he tries to access a page that

requires him to be logged in. The check_path is where the user will be redirected when he tries to login. The logout path is where the user will be redirected when he logs out. app_login and app_logout are the name of the routes for the login and logout functions. For the login, it is in LoginController.php
#[Route('/login', name: 'app_login', methods: ['GET', 'POST'])]
public function index(AuthenticationUtils $authenticationUtils): Response

access_control: These are rules for restricting access based on URL patterns and roles. The commented-out examples show how you might restrict access to certain paths based on user roles.
For instance, # - { path: ^/admin, roles: ROLE_ADMIN } means that you would have to have the role ROLE_ADMIN to access the /admin path. If you don't have this role, you will be redirected to the login page.

when@test:

This is used to configure the security for the test environment. The parameter here is the password hashing options which are less secure than in the prod environment. This is because we don't want to waste time and ressources hashing the password in the test environment.

Advice for beginners
1. Be careful about front and back
if you need to make a condition for a specific action, such as something only admins can do, make sure that you have a check both in the twig template AND the php logic. Just because I cannot see the button in the twig template doesn't mean I cannot access the page by typing the url in the browser. You need to make sure that the php logic is also protected, because I can still access the page by typing the url in the browser or by using postman.

2. Test your code
You might be tempted to just make your code work and call it a day, but it is very important to implement unit tests and functional tests. This will allow you to make sure that your code is working as intended, and that you don't break anything when you make changes. It is also a good way to make sure that your code is secure and then refactor it if needed. In short, if the tests don't pass, it doesn't "truly" work even if it does in the browser.