ToDo on Symfony 6 Framework : Documentation

I. Application Security

The Symfony Framework has been perfected since its creation more than a decade ago. Aware of developer’s needs in a business environment, the security component is quite simple to apprehend and managing one site with it ensures the application to have a security standards compliant with the current norms.

The security bundle has been installed with all its dependencies on our project. Let’s see where to find its various elements and what settings we can implement as a developper using Symfony.

1. Account management :

An account is incarnated by a User entity (namespace : App\Entity) . Symfony expects the user entity to contain key information that he will use for authentication :

- Do not delete the files the user entity implements . It allows symfony to identify the information needed.
  - UserInterface :
    - the user identifier : its the string attributes that will be used as a unique nametag for a user (the current one is the email). With this identifier, Symfony can retrieve the entity to compare information entered by a user and confirm an authentication. It also can ensure two different users cannot share the same identifier.
    - The Roles attribute. A json attribute that will describe the users right through established roles.
  - PasswordAuthenticatedUserInterface.
    - Its annotation will designate the attribute to be used as a password.
    -

You can change what will be used as unique identifier and password. Make sure the user identifier is unique for each users.

The roles attribute should not be changed. It is a convention found in all websites.

2. Authentitcation :

First, let us check what the hidden dependencies do for us. What we do not see : The user provider loads (and reloads at the beginning of each request) users from any storage (e.g. the database) based on a "user identifier". At the beginning of each request, the user is loaded from the session.

What we see : The authenticator must pass the necessary information to the user provider so it can validate the account exists and give access to its rights. The authenticator read the user identifier and password, creates a passport, a symfony object to pass on to the provider. The user provider to load a corresponding user, thus finding a match. If finding a match is a success, authentication is a success.

- UserAuthenticator.php (namespace : App\Security) : you’ll see the creation of the passport on the authenticate function.
  - **You can change the page the user is redirected to during the initial login. Just change the redirection’s route name under the onAuthenticationSuccess.**
  - **Note that you can create other authenticators to add other security checks you might want to add.**

- RegistrationController.php (namespace : App\Controller) : A controller dedicated to the creation of new accounts. **If you want to add logic to this functionality, this is where you will find the logic.**
- LoginController.php (namespace : App\Controller) : A controller dedicated to the login route. But it only declares its existence. The logic is handled through the Symfony authenticator. If needed, Symfony documentation tells you how to override this.
- **You can also adapt the form Login template in the templates**

3. Roles and rights :

In security.yaml, under access_control, we set up our rules of access depending on rights and url: **An important and easy way to manage security access if we can sum-up the access rules under url hierarchies**. For example, putting ROLE_ADMIN in front of ^/users tells symfony every url with /user are exclusively dedicated to the Admin role. We could create a super_admin role that as the exclusive rights to delete user by adding "*- { path: ^/users/delete, roles: ROLE_SUPER_ADMIN }*" thus excluding other admin for this action.

If you cant handle access right through simple url regroups, the access rule must be implemented in controllers with rest of the business logic's algorithms using the user session or in the twig templates to display some options. You can access the session with :

- $this→getUser() in controllers
- app.user in twig templates

4. Security.yaml :

Security.yaml (namespace : config\packages), Configures our symfony security. We'll find elements of the various files we have just seen :

- The User class is annotated under the app_user_provider. The property annotation will designate our user identifier. **That is how we the provider which entity incarnate an account within our application and what is its identifier.**
- Under firewalls, we designate :
  - the user provider,
  - the authenticator(s) to use,
  - the login and logout routes
  - **the redirection to implement after logout**
- Under access_control, the url access rules as seen in 3.

II. General guidelines

ToDo integrates the Symfony 6 framework and follows its principles. **To updates its functionalities**, you must respect the architecture of the site to keep the source code intelligible and accessible by any developper.

To add attributes to an entity, you can either use the maker command from the maker bundle (link) or adapt yourself the entity file, repository and formType (follow the existing models of the application).
In the same spirit, **to create a new entity**, you can either use the maker command from the maker bundle or create yourself the entity file, repository and formType (follow the existing models of the application).

Business logic (namespace : App/Controller): any **new functionality** our **update of functionalities** must be implemented on distinct controllers. In its current state, ToDo app is animated by two entities: Task and User. The Task functionalities must be written in the TaskController, the User functionalities must be written in the UserController. **Make sure to annotate any new function with an url and a name to make it known to the router and comment them to sumup what it does.**

Repository (namespace: App/Repository): In the controllers, you have access to doctrine methods to access your data. If the methods are not enough to get the data sample you want, **specific requests must be written**. Following the controller's example, Users and Tasks have their own repository. **If you want to retrieve Tasks, write the logic in the TaskRepository (user in UserRepository)**. Repositories are accessible from the controllers. Follow the syntax : $this->em->getRepository(ClassWanted::class)->methodNeeded();

The only logic that escapes those rules are the security specifications described in the first part with specific controllers.

Good luck and good development !!!!!