ToDo & Co - Authentication

Introduction

The application is based on the Symfony framework, and uses the Security component for the authentication part.

This documentation explains how the authentication process works inside the ToDo & Co application, where are the configuration files and which files modify to make changes to authentication.

1. Authentication

In Symfony, the authentication process needs a *User class*, no matter how the users will authenticate, or where the data will be stored.

1.1. The User Class

In Symfony, a user is represented by a User entity. This entity will hold the data needed to identify the user (username, email, password, etc.).

- This class must implement `UserInterface`.
- The *User class* is a Doctrine entity, and in this project all the User's data is stored in the database, in the `user` table.
- An attribute from the *User class* is needed to uniquely identify a user, which is the **email** in this project. This parameter can be modified in `config/packages/security.yaml`, under the `providers` key.
- In order to enable Symfony to check for user uniqueness (when validating forms for user creation/update for example), the `@UniqueEntity("email")` annotation must be present in the *User class*. In Symfony 5.3, a new method `getUserIdentifier()` is used to get the unique identifier for the user. In this project, this method returns the user's **email**.
- For a security purpose, the user's **password** must be hashed before it's stored in the database. To enable password hashing in Symfony, under the `password_hashers` key in `security.yaml`, an `algorithm` key must be defined. By default, the `auto` option is set, which lets Symfony choose the best algorithm available (in Symfony 5.3 the algorithm is *bcrypt*).
- When password hashing is needed in the code, e.g. when creating a new user, simply inject the `UserPasswordHasherInterface` as a service with dependency injection, and then use it like this :

```
$hashedPassword = $passwordHasher->hashPassword($user, $rawPassword);
```

1.2. Firewall

User authentication in Symfony is managed by a firewall. The configuration of this firewall defines how the user will be authenticated (login form, API token, etc.), and to which parts of the application the user has access.

The firewall configuration is available in `security.yaml`, under the `firewalls` key.

In this project, the `main` firewall defines that all the URLs of the site needs an authentication (`pattern` key) via the login form (`entry_point` key), e.g. all unauthenticated users will be redirected to the login page, to authenticate and have access to these URLs.

The `SecurityController` is in charge of rendering the login page, which contains the login form, and also to get the last authentication error (if wrong credentials were provided by the user).

Then, when the login form is submitted, authentication will be handled by Symfony's *Authenticators*, so no other action is needed to authenticate users.

2. Authorization

In order to restrict users access to some parts of the site, e.g. the `admin` section, an authorization process will be used.

This process is based on the `roles` of the user. Inside the *User class*, the `getRoles()` method retrieves the user's roles stored in the database. If the user has no role, a default `ROLE_USER` is added automatically.

For this project another role has been added, the role `ROLE_ADMIN`. This role grants access to all the URLs starting by `/users` (users creation, update and deletion) to the users who have this role.

If needed, it's possible to add new roles when creating/updating users, the only mandatory rule is that a role **must** start with `ROLE_`. After that part, any string will be valid (e.g. `ROLE_TASK_UPDATE`).

To deny access to some URLs of the site, there are two ways :

- the `access_control` key in `security.yaml`
- in `Controllers` or in a *Twig* template

For this project, the two options have been used, depending on the needs.

The `access_control` key in `security.yaml` is used to grant access to users who have the `ROLE_ADMIN` role to all URLs starting by `/users`, via the `- { path: ^/users, roles: ROLE_ADMIN }` option.

In the code, it's possible to use `$this->denyAccessUnlessGranted(<role>)` in a Controller, to deny access to a route for example.

Or, in a Twig template it's possible to use :

```
{% if is_granted('ROLE_ADMIN') %}
  <a href="...">Supprimer</a>
{% endif %}
```

to show a link only if the user has the role `ROLE_ADMIN`

No matter the way used, if the user does not have access to the section of the site, a 403 HTTP error will be sent.

Learn more

- The Security documentation
- The Security component
- The Authentication part of the security component

Authentication process