AUTHENTICATION IMPLEMENTATION GUIDE

Amélie-Dzovinar Haladjian

The authentication system for Todo&Co's to-do list application was implemented using Symfony's Guard Authenticator.

1. User Entity

First of all, the Entity/User needs to implement the Symfony/Component/Security/Core/User/UserInterface. This interface contains 5 methods such as getRoles, getUsername and getPassword that must be defined in our Entity/User class in order to make sure that our authentication system works.

```php
namespace App\Entity\User;

use App\Entity\Security\Roles;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Table("user")
 * @ORM\Entity(repositoryClass="App\Repository\User\UserRepository")
 * @UniqueEntity("email")
 * @UniqueEntity("username")
 */
class User implements UserInterface
{
```

2. Authenticator

The AppAuthenticator class was generated using the command bin/console make:auth.

This class extends Symfony's AbstractFormLoginAuthenticator and implements 6 main methods.

```php
public function supports(Request $request)
{
    return 'login' === $request->attributes->get( key: '_route')
        && $request->isMethod( method: 'POST');
}
```

This method is used to determine whether the authenticator supports the given request. In this case, the authenticator will be executed if the method POST is used on our login route.

```php
public function getCredentials(Request $request)
{
    $credentials = [
        self::USERNAME => $request->request->get( key: self::USERNAME),
        self::PASSWORD => $request->request->get( key: self::PASSWORD),
        'csrf_token'   => $request->request->get( key: '_csrf_token'),
    ];
    $request->getSession()->set(
        Security::LAST_USERNAME,
        $credentials[self::USERNAME]
    );

    return $credentials;
}
```

The method getCredentials returns the credentials submitted by the user through the login form. They will then be used to check whether the credentials are valid and retrieve the corresponding user from the database.

```php
public function getUser($credentials, UserProviderInterface $userProvider)
{
    $token = new CsrfToken( id: 'authenticate', $credentials['csrf_token']);
    if (!$this->csrfTokenManager->isTokenValid($token)) {
        throw new InvalidCsrfTokenException();
    }

    $user = $this->entityManager->getRepository( className: User::class)->findOneBy(
        [self::USERNAME => $credentials[self::USERNAME]]
    );

    if (!$user) {
        throw new CustomUserMessageAuthenticationException( message: 'Ce nom d\'utilisateur n\'existe pas');
    }

    return $user;
}
```

If the username retrieved from the getCredentials method matches a user from the database, this method will return it. Otherwise, it will throw an exception that will then be displayed on the login form.

```php
public function checkCredentials($credentials, UserInterface $user)
{
    return $this->passwordEncoder->isPasswordValid($user, $credentials[self::PASSWORD]);
}
```

This method is used to check whether the password from the request matches the password from the user returned by the getUser method.

```php
public function onAuthenticationSuccess(Request $request, TokenInterface $token, $providerKey)
{
    $targetPath = $this->getTargetPath($request->getSession(), $providerKey);

    if ($targetPath) {
        return new RedirectResponse($targetPath);
    }

    return new RedirectResponse($this->urlGenerator->generate( name: 'list_tasks'));
}
```

This method defines the app's behavior once the user has successfully logged in. In this case, we redirect the user to the task list or to the page he tried to access before being invited to log in.

```php
protected function getLoginUrl()
{
    return $this->urlGenerator->generate( name: 'login');
}
```

This last method is called in the start and onAuthenticationFailure methods from AbstractFormLoginAuthenticator and returns the route the user should be redirected to if the user is not authenticated or if the authentication fails.

3. Security configuration

The app security is configured inside config/packages/security.yml.

```yaml
security:
    encoders:
        App\Entity\User\User: bcrypt
```

Encoders are used to indicate which algorithm should be used to encode user passwords. The authenticator will use this encoder when checking whether the password from the request matches that of the user retrieved from the database.

```
providers:
    doctrine:
        entity:
            class: App\Entity\User\User
            property: username
```

This section is used to determine how users are loaded during authentication. In this case, we are loading users from the database.

```
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        anonymous: true
        logout: ~
        guard:
            authenticators:
                - App\Security\AppAuthenticator
```

Firewalls are used to determine how users will authenticate.

The dev firewall is used to prevent any tools used to debug in dev environment from being blocked by our security system.

The main firewall is used to configure our app's main authentication system. In this case, we are using our own guard AppAuthenticator.

```
access_control:
    - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
    - { path: ^/admin, roles: ROLE_ADMIN }
    - { path: ^/, roles: ROLE_USER }
```

The last section is used to control user access to different parts of our app. In this case, users need to be authenticated to access any part of our app aside from the login page, and authenticated users need to have the role ROLE_ADMIN in order to access the admin panel.