To Do List App

Authentication documentation

Created by	Dubreucq Alexandra
To	ToDo&Co OpenClassRooms
Creation date	12/04/2018

Sommaire

Introduction

The goal of this documentation is to learn to a beginner with the Symfony framework :

- Which file(s) to edit and why
- How authentication works
- And where are the users stored

Specific authorization is also explained.

Symfony Security Component

The Security component is powerful and provides a complete security system for your web applications.
There are two distincts notions to really understand before going further:
- Authorization
- Authentication

Authentication

Authentication is the process managed by firewalls which defines who you are as user.
For the application you're an anonymous user or an authenticated user and then a logged in member.
Authentication allows you to secure some parts of your website by forcing anonymous user to log in as a member. If user is logged in as a member, firewall lets him pass, else anonymous user will be redirected to authentication page.

Authorization

Authorization is the process managed by access control which defines if you are allowed to access to the requested resource (e.g. a page).
For example you may have an admin area only accessible by users with administrator rights so you don't want any other type of user to access this section. Access Control will check if authenticated user has authorization to access resource.
Read more about Security Component on [official documentation][1]

How it works

The security system is configured in 'app\config\security.yml' file.

Let's decrypt all its content and parameters before going further.

- **encoders**: allows you to choose the encoding type you want for your users' password, such as `bcrypt` or `sha12` for example.
- **role_hierarchy**: allows you to define different roles for your users.
- **providers**: allows you to define from where you want to get your users. Providers are the users providers used by firewalls to get users and identify them.
- **firewalls**: allows you to define several firewalls you want to use to secure parts of your application.
- **access_control**: allows you to define the routes you want to restrict to specific roles.

Attention: firewalls order is important because the first corresponding to the requested route is the used one.

How it is done

In our application we needed to :

1. restrict all website pages to members ;
2. allow member to be able to create, update and delete his own tasks ;
3. restrict users management area (create, update, delete members) to administrators.

To achieve this we had to define roles to users so we will be able to authorize/unauthorize them to access to resources by checking their role.

Now let's analyze how it has been done in our `app\config\security.yml` file.

```
security:
```

Encoders

In our case the chosen encoding method is `bcrypt` for our entity `AppBundle\Entity\User`.

```
security:
    encoders:
        AppBundle\Entity\User: bcrypt
```

Roles Hierarchy

We have implemented two roles:

```
role_hierarchy:
    ROLE_USER: ROLE_USER
    ROLE_ADMIN: [ROLE_USER, ROLE_ADMIN]
```

As you can see in the above configuration, a user with `ROLE_ADMIN` also have the `ROLE_USER` role. Think about hierarchy logic!

Attention: all roles you assign to a user must begin with the `ROLE_` prefix!

Note: In our application, an admin user may manage users and assign them roles. In `Form\UserType`, roles are retrieved dynamically from `security.role_hierarchy.roles` (by a service to inject properly dependency). That way we still have to edit only one file when we want to edit roles.

Providers

Our users are stored in database so we have to define a Doctrine provider.

```
providers:
    doctrine:
        entity:
            class: AppBundle:User
            property: username
```

Here you can see that users are of `AppBundle:User` type and loaded by their `username` property.

Firewalls

The `dev` firewall isn't important. It only ensures that Symfony's development tool works fine. Symfony's development tool lives under `/_profiler` and `/_wdt` URL's.

The `main` firewall will handle all the other URL's. The pattern `^/` value means this firewall has to handle all URL's (except those defined previously if they're catched before).

We can see that all our website pages are restricted to authenticated users behind our `main` firewall:

```
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false

    main:
        anonymous: ~
        pattern: ^/
        form_login:
            login_path: login
            check_path: login_check
            always_use_default_target_path:  true
            default_target_path:  /
        logout: ~
```

If you think of the firewall like your security system, it makes sense to have just one main firewall. But this does not mean that every URL requires authentication, the `anonymous` key takes care of this. In fact as long as you don't log in as a website member, you're "authenticated" as `anon.`. You're just an anonymous user.

Users will authenticate with a login form so we enable `form_login` under our `main` firewall. We give to `login_path` the route to access to this form. The route `/login` will call the loginAction method in `Controller/SecurityController` which will render the form view.

```
/**
 * @Route("/login", name="login")
 */
public function loginAction()
{
    // login action
    // render view
}
```

Firewall will automatically intercept and transform any submitted form to the `/login_check` URL so we don't have to implement a controller or method for this URL.

The `always_use_default_target` path ignore the previously requested URL and will always redirect to `default_target_path` after any successful login. So if you want redirect user to the previous requested URL after he successfully logged in, just disable the `always_use_default_target` parameter and remove the `default_target_path` one.

By default the `logout` key is not configured. When users log out from any firewall their sessions become invalid.

Access Control

The most basic way to secure parts of your application is to secure an entire URL pattern.

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

In the above configuration, you can see that the `/` path requires `ROLE_USER` role. It means that all the application's URL's need that user has at least `ROLE_USER` role if he does not want to be redirected on login page.

The `IS_AUTHENTICATED_ANONYMOUSLY` role means "all users", since we saw that you're "authenticated" as an anonymous user until you log in as a member. Here, we authorize the `/login` path to be accessible by anonymous, otherwise users will never be able to log in to our application and we will generate an infinite loop. Please be attentive to the order of controls: if we had defined the `^/` path first, the `/login` path would always be intercepted by the first configuration.

We could have define our users management area security in `access_control` like:

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

But since Symfony documentation tells in [security best practices][2]:

> *Whenever possible, use the `@Security` annotation.*

We used the `@Security` annotation in our `UserController` to secure all its methods!

```
/**
 * @Security("has_role('ROLE_ADMIN')")
 */
class UserController extends Controller
{
    // UserController methods
}
```

Authorization

As explained above, we check users' authorization to access resources according to their roles. But for our "*allow member to be able to create, update and delete his own tasks*" need (see [How it is done][3]), it is managed by a [voter][4].

Security voters are the most granular way of checking permissions (e.g. "can this specific user edit the given item?").

In our `Security\TaskVoter` file you will find the conditions to check user permissions for tasks edition and deletion. That way we are able to define `edit` and `delete` buttons visibility in views :

```
{% if is_granted('edit', task) %}
    <!-- edit button -->
{% endif %}
{% if is_granted('delete', task) %}
    <!-- delete button -->
{% endif %}
```

And in controllers (`TaskController` in this case), we are able to check user access to the `edit` and `delete` actions:

```
/**
 * @Route("/tasks/{id}/edit", name="task_edit")
 */
public function editAction(Task $task, Request $request)
{
    $this->denyAccessUnlessGranted('edit', $task);
    // ... all the edit action
}

/**
 * @Route("/tasks/{id}/delete", name="task_delete")
 */
public function deleteTaskAction(Task $task)
{
    $this->denyAccessUnlessGranted('delete', $task);
    // ... all the delete action
}
```

Conclusion

With all those informations you are now able to manage our application security, add another role, restrict another controller or path to specific role, edit user permissions to specific actions, ...

Thanks for reading and enjoy!

[1]: https://symfony.com/doc/3.4/components/security.html
[2]: https://symfony.com/doc/3.4/best_practices/security.html#authorization-i-e-denying-access
[3]: https://github.com/bhalexx/todo-and-co/blob/master/documentation/Authentication.md#how-it-is-done
[4]: https://symfony.com/doc/3.4/security/voters.html