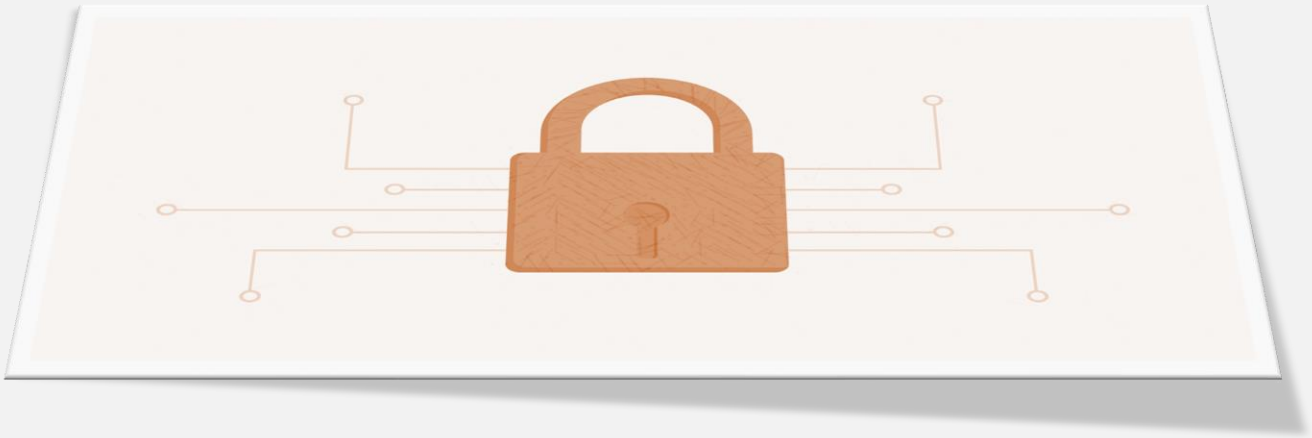

Technical Documentation: Authentication Guide

20 AOUT 2023

MEHDI HADDOU | DAPS – P8

Authentication Guide

I. INTRODUCTION

The purpose of this documentation is to guide future junior developers in understanding how authentication has been implemented in the Symfony 5.4 project. It explains which files have been modified, how authentication works, where user information is stored, and how to ensure smooth collaboration and code quality maintenance.

II. MODIFIED FILE

To implement authentication, the file *config/packages/security.yaml* has been modified. This file contains the security configuration of the application, where parameters related to authentication and access roles are defined.

III. AUTHENTICATION PROCESS

The authentication process in the Symfony application is managed by the integrated security component. Here's how it is configured in the file *config/packages/security.yaml* :

```yaml
# config/packages/security.yaml
security:
    enable_authenticator_manager: true
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
    providers:
        app_user_provider:
            entity:
                class: App\Entity\User
                property: username
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: app_user_provider
            custom_authenticator: App\Security\SecurityAuthenticator
            logout:
                path: logout
                target: homepage

    access_control:
        - { path: ^/login, roles: PUBLIC_ACCESS }
        - { path: ^/users, roles: ROLE_ADMIN }
        - { path: ^/coverage, roles: PUBLIC_ACCESS }
        - { path: ^/, roles: IS_AUTHENTICATED_FULLY }
```

Explanation:

```
enable_authenticator_manager: true
```

« enable_authenticator_manager » instructs Symfony to use the authentication manager.

```
password_hashers:
    Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
```

« password_hashers » configures password hashing.

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

« providers » specifies the user provider for authentication. In this case, "app_user_provider" uses the "User" entity as the user class and the "username" property as the login identifier.

```
firewalls:
    dev:
        pattern: ^/(_(profiler|wdt)|css|images|js)/
        security: false
    main:
        lazy: true
        provider: app_user_provider
        custom_authenticator: App\Security\SecurityAuthenticator
        logout:
            path: logout
            target: homepage
```

« firewalls » define security firewalls for different paths of the application. In this example, "main" is the main firewall used for authentication:

- « lazy » indicates that authentication occurs only when needed, improving performance ;
- « provider » user provider defined earlier in our configuration ;
- « custom_authenticator » specifies the custom class "SecurityAuthenticator" that handles the authentication process. This class performs necessary checks and authenticates users.
- « logout » configures the logout path and the redirection after logout to the route defined in "target".

```yaml
access_control:
    - { path: ^/login, roles: PUBLIC_ACCESS }
    - { path: ^/users, roles: ROLE_ADMIN }
    - { path: ^/coverage, roles: PUBLIC_ACCESS }
    - { path: ^/, roles: IS_AUTHENTICATED_FULLY }
```

« access_control » defines access rules based on user roles for different paths of the application. It's crucial to note that the order of rules in "access_control" matters. Rules are evaluated in the order they are defined, and the first matching rule is applied. Hence, it's essential to define the most specific rules first, followed by more general rules. If a more general rule is defined before a specific rule, it might override the intended restrictions. This configuration controls access to different parts of the application based on user roles and ensures proper security.

IV. USER STORAGE

User information is stored in the database and managed by Symfony's Object-Relational Mapping (ORM) system, Doctrine. Users are represented by the class "App\Entity\User," which is an entity defined in the project. This class defines the structure and properties of users, such as the username, hashed password, and roles.

Doctrine handles the creation, updating, and retrieval of user data from the database. Through the configuration of the user provider in the "security.yaml" file, Symfony can interact with Doctrine to authenticate users during the authentication process.

In summary, users are stored in the database and managed by Doctrine, and the class "App\Entity\User" represents the user structure in the Symfony project.

V. CONCLUSION

This technical documentation has covered the essential aspects of implementing authentication in the Symfony 5.4 application. We have explored the different stages of the authentication process, highlighting the key files and configurations necessary for its proper functioning.

Furthermore, we delved into user storage in the database using the "App\Entity\User" entity. This entity is managed by Doctrine and provides a structure to store user-related information, including their identifiers and roles. We have also emphasized the importance of adhering to security rules, particularly the order of access rules in "access_control," to ensure appropriate access to different parts of the application based on user roles.

It is important to note that this documentation is intended to facilitate the understanding and onboarding of the authentication process for junior developers joining the team. It provides detailed explanations of key steps, configurations, and elements involved in the authentication flow. This will enable them to confidently navigate through configurations, classes, and key concepts related to authentication while ensuring the security and efficiency of the application.