Actividad

Symfony. Seguridad. La clase usuario. Autenticación

1. Seguridad

Symfony proporciona muchas herramientas para proteger tu aplicación. Algunas herramientas de seguridad relacionadas con HTTP, como las cookies de sesión segura y la protección CSRF, se proporcionan de forma predeterminada.

El paquete **SecurityBundle, proporciona todas las funciones de autenticación y autorización necesarias para proteger la aplicación**.

Para comenzar, instale SecurityBundle:

```
composer require symfony/security-bundle
```

Si tienes Symfony Flex instalado, esto también crea un archivo de configuración security.yaml para ti:

```yaml
# config/packages/security.yaml
security:
    # https://symfony.com/doc/current/security.html#registering-the-user-hashing-passwords
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface: 'auto'
    # https://symfony.com/doc/current/security.html#where-do-users-come-from-user-providers
    providers:
        users_in_memory: { memory: null }
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: users_in_memory

            # activate different ways to authenticate
            # https://symfony.com/doc/current/security.html#firewalls-authentication

            # https://symfony.com/doc/current/security/impersonating_user.html
            # switch_user: true

    # Easy way to control access for large sections of your site
    # Note: Only the *first* access control that matches will be used
    access_control:
        # - { path: ^/admin, roles: ROLE_ADMIN }
        # - { path: ^/profile, roles: ROLE_USER }
```

los tres elementos principales de toda configuración de seguridad son:

Usuarios (proveedores)

Cualquier sección segura de la aplicación necesita algún concepto de usuario. El proveedor de usuarios carga a los usuarios desde cualquier almacenamiento (por ejemplo, la base de datos) en función de un "identificador de usuario" (por ejemplo, la dirección de correo electrónico del usuario);

El cortafuegos y la autenticación de usuarios (cortafuegos)

El firewall es el núcleo de la protección de la aplicación. Cada solicitud dentro del firewall se verifica si necesita un usuario autenticado. El cortafuegos también se encarga de autenticar a este usuario (por ejemplo, mediante un formulario de inicio de sesión);

Control de acceso (autorización) (access_control)

Con el control de acceso y el comprobador de autorizaciones, puede controlar los permisos necesarios para realizar una acción específica o visitar una URL específica.

2. El usuario (la clase usuario)

Los permisos en Symfony siempre están vinculados a un objeto de usuario. Si necesitas proteger (partes de) la aplicación, **debes crear una clase de usuario**. Esta clase debe implementar la interfaz UserInterface. **A menudo esta clase se trata de una entidad de Doctrine, pero también puede utilizar una clase de usuario de seguridad dedicada.**

La forma más fácil de generar una clase de usuario es usando el comando make:user de MakerBundle:

```
php bin/console make:user
 The name of the security user class (e.g. User) [User]:
 > Usuario

 Do you want to store user data in the database (via Doctrine)? (yes/no) [yes]:
 > yes

 Enter a property name that will be the unique "display" name for the user (e.g. email, username, uuid) [email]:
 > email

 Will this app need to hash/check user passwords? Choose No if passwords are not needed or will be checked/hashed by some other system (e.g. a single sign-on server).

 Does this app need to hash/check user passwords? (yes/no) [yes]:
 > yes

 created: src/Entity/User.php
 created: src/Repository/UserRepository.php
 updated: src/Entity/User.php
 updated: config/packages/security.yaml
```

la clase Usuario implementará determinadas características de seguridad determinadas

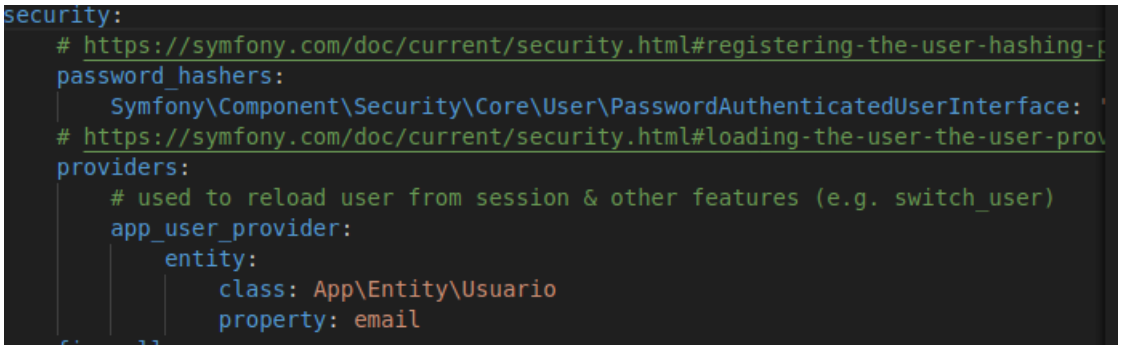
```php
<?php

namespace App\Entity;

use App\Repository\UsuarioRepository;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
use Symfony\Component\Security\Core\User\UserInterface;

#[ORM\Entity(repositoryClass: UsuarioRepository::class)]
class Usuario implements UserInterface, PasswordAuthenticatedUserInterface
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 180, unique: true)]
    private ?string $email = null;

    #[ORM\Column]
    private array $roles = [];

    /**
     * @var string The hashed password
     */
    #[ORM\Column]
    private ?string $password = null;

    public function getId(): ?int
    {
        return $this->id;
    }
```

Si el usuario es una entidad de Doctrine, crea las tablas mediante los siguientes comandos, como hemos hecho antes:

```
php bin/console make:migration
php bin/console doctrine:migrations:migrate
```

3. Cargando el usuario: El proveedor de usuarios

Además de crear la entidad, el comando make:user también agrega configuración (carpeta config) para un proveedor de usuarios en la configuración de seguridad:



Este proveedor de usuarios sabe cómo (re)cargar usuarios desde un almacenamiento (por ejemplo, una base de datos) basándose en un "identificador de usuario" (por ejemplo, la dirección de correo electrónico o el nombre de usuario del usuario). La configuración anterior utiliza Doctrine para cargar la entidad User utilizando la propiedad email como "identificador de usuario".

Los proveedores de usuarios se utilizan en dos sitios durante el ciclo de vida de la seguridad:

Cargar el usuario en función de un identificador

> **Durante el inicio de sesión** (o cualquier otro autenticador), **el proveedor carga el usuario en función del identificador de usuario**. Algunas otras funciones, como la suplantación de usuario y Remember Me también usan esto.

Recarga del usuario desde la sesión

> **Al principio de cada solicitud, el usuario se carga desde la sesión** (a menos que el firewall no tenga estado). El proveedor "refresca" al usuario (por ejemplo, se vuelve a consultar la base de datos para obtener datos nuevos) para asegurarse de que toda la información del usuario esté actualizada (si es necesario, se anula la autenticación del usuario o se cierra la sesión si algo cambia.

Symfony viene con varios proveedores de usuario integrados:

a) **Proveedor de usuarios de entidad (el que usaremos)**
   - Carga usuarios desde una base de datos usando Doctrine;

b) **Proveedor de usuarios LDAP**
   - Carga usuarios desde un servidor LDAP;

c) **Proveedor de usuarios de memoria**
   - Carga usuarios desde un archivo de configuración;

d) **Proveedor de usuarios de la cadena**
   - Combina dos o más proveedores de usuarios en un nuevo proveedor de usuarios. Dado que cada firewall tiene exactamente *un* proveedor de usuario, puede usarlo para encadenar varios proveedores.

Proveedor de usuarios de entidad (el que usaremos)

Este es el proveedor de usuarios más común. Los usuarios se almacenan en una base de datos y el proveedor de usuarios utiliza Doctrine para recuperarlos.

```yaml
# config/packages/security.yaml
security:
  providers:
    users:
      entity:
        # the class of the entity that represents users
        class: 'App\Entity\User'
        # the property to query by - e.g. email, username, etc
        property: 'email'

        # optional: if you're using multiple Doctrine entity
        # managers, this option defines which one to use
        #manager_name: 'customer'

  # ...
```

4. Usar una consulta personalizada para cargar u usuario

El proveedor de entidades solo puede consultar por un *campo específico*, especificado por la clave de configuración de la propiedad (en nuestro caso el email) Si deseas un poco más de control sobre esto, por ejemplo, si deseas encontrar un usuario por correo electrónico *o* nombre de usuario, puede hacerlo implementando UserLoaderInterface en su repositorio de Doctrine (por ejemplo, UsuarioRepository

Crea el siguiente método en la clase UsuarioRepository para que sea el método personalizado de carga de un usuario:

```php
public function loadUserByIdentifier(string $usernameOrEmail): ?User
    {
        $entityManager = $this->getEntityManager();

        return $entityManager->createQuery(
                'SELECT u
                FROM App\Entity\Usuario u
                WHERE u.username = :query
                OR u.email = :query'
            )
            ->setParameter('query', $usernameOrEmail)
            ->getOneOrNullResult();
    }
```

Para finalizar, quita la clave de propiedad del proveedor de usuarios en security.yaml:

```yaml
# config/packages/security.yaml
security:
  providers:
    users:
      entity:
        class: App\Entity\User

  # ...
```

IMPORTANTE: Ahora, cada vez que Symfony utilice el proveedor de usuarios, se llamará al método loadUserByIdentifier() en su UserRepository.

5. Registro del usuario: Hashing de contraseñas

Muchas aplicaciones requieren que el usuario inicie sesión con una contraseña. Para estas aplicaciones, el paquete SecurityBundle proporciona funciones de verificación y hash de contraseñas.

En primer lugar, asegúrate que la clase Usuario implementa PasswordAuthenticatedUserInterface:

Vemos que **en nuestro caso ya se ha generó la clase que implementa esta interfaz**, luego estará todo correct:o

En el archivo de configuración de seguridad de haberse establecido el método de generar HASH para las contraseñas. Compruébalo

```yaml
when@test:
    security:
        password_hashers:
            # By default, password hashers are resource intensive and take time. This is
            # important to generate secure password hashes. In tests however, secure hashes
            # are not important, waste resources and increase test times. The following
            # reduces the work factor to the lowest possible values.
            Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface:
                algorithm: auto
                cost: 4 # Lowest possible value for bcrypt
                time_cost: 3 # Lowest possible value for argon
                memory_cost: 10 # Lowest possible value for argon
```

IMPORTANTE : Ahora que Symfony sabe *cómo* quieres hacer hashs de las contraseñas, puedes usar el servicio UserPasswordHasherInterface para hacerlo antes de guardar tus usuarios en la base de datos:

Crea un **controlador de registro (RegsitroController) cuya lógica sea la de** crear un nuevo usuario registrado con la contraseña protegida con hash de la siguiente forma.

NOTA: Este controlador de registro será cambiado más adelante por otro

```php
#[Route('/registro', name: 'app_registro')]
public function registrar(UserPasswordHasherInterface $passwordHasher,
EntityManagerInterface $entityManager): Response
{
    // NOTA suponemos que estos datos vienen de un formulari HTML
    $usuario= new Usuario();
    $usuario->setEmail('user1@gmail.com');
    $passwordTexto = '1234';

    // hash the password (based on the security.yaml config for the $user class)
    $hashedPassword = $passwordHasher->hashPassword(
        $usuario,
        $passwordTexto
    );
    $usuario->setPassword($hashedPassword);
    $entityManager->persist($usuario);
    $entityManager->flush();
    return $this->render('registro/index.html.twig', [
        'controller_name' => 'RegistroController',
    ]);

}
```

Comprueba el resultado de la base de datos:

←T→		id	email	roles (DC2Type:json)	password
☐	Editar Copiar Borrar	8	user1@gmail.com	[]	$2y$13$tfTd0JuTMRSSI3t/q3lhHugFM9W6HPFyVbnNkE.lDgf...

IMPORTANTE: Si la clase Usuario es una entidad Doctrine y se aplican hash a las contraseñas de usuario, la clase de repositorio de Doctrine relacionada con la clase de usuario debe implementar PasswordUpgraderInterface.

El comando make:registration-form maker puede ayudarte a configurar el controlador de registro y añadir funciones como la verificación de la dirección de correo electrónico mediante SymfonyCastsVerifyEmailBundle.

Instala los **siguientes validadores de formularios**

```
daw@daw-VirtualBox:~/my_project_directory$ composer require symfonycasts/verify-email-bundle
```

```
daw@daw-VirtualBox:~/my_project_directory$ composer require validator
```

Vamos a crear un formulario de registro con su controlador. Para ello ejecuta la siguiente secuencia de comandos:

```
daw@daw-VirtualBox:~/my_project_directory$ php bin/console make:registration-form
 Creating a registration form for App\Entity\Usuario

 Do you want to add a #[UniqueEntity] validation attribute to your Usuario class to make sure duplicate accounts aren't created?
s/no) [yes]:
 > no

 Do you want to send an email to verify the user's email address after registration? (yes/no) [yes]:
 > no

 Do you want to automatically authenticate the user after registration? (yes/no) [yes]:
 > yes

 ! [NOTE] No Guard authenticators found - so your user won't be automatically
 !        authenticated after registering.

 What route should the user be redirected to after registration?:
  [0 ] _preview_error
  [1 ] create_product
```

NOTA: En este punto no te preocupes de la redirección, selecciona cualquier opción, luego la cambiaremos

```
created: src/Form/RegistrationFormType.php
created: src/Controller/RegistrationController.php
created: templates/registration/register.html.twig

 Success!

Next:
Make any changes you need to the form, controller & template.

Then open your browser, go to "/register" and enjoy your new form!
```

Fíjate que como este comando te ha creado un controlador con toda la lógica para crear un formulario , procesarlo y dar de alta al nuevo usuario en la base de datos

```php
class RegistrationController extends AbstractController
{
    #[Route('/register', name: 'app_register')]
    public function register(Request $request, UserPasswordHasherInterface $userPasswordHasher,
    EntityManagerInterface $entityManager): Response
    {
        $user = new Usuario();
        $form = $this->createForm(RegistrationFormType::class, $user);
        $form->handleRequest($request);

        if ($form->isSubmitted() && $form->isValid()) {
            // encode the plain password
            $user->setPassword(
                $userPasswordHasher->hashPassword(
                    $user,
                    $form->get('plainPassword')->getData()
                )
            );

            $entityManager->persist($user);
            $entityManager->flush();
            // do anything else you need here, like send an email

            //return $this->redirectToRoute('tarea_show');


            return new Response(
                'Regsitro creado'
            );


        }
        return $this->render('registration/register.html.twig', [
            'registrationForm' => $form->createView(),
        ]);
    }
}
```

6. Formulario de inicio de sesión (Login)

La mayoría de los sitios web tienen un formulario de inicio de sesión en el que los usuarios se autentican mediante un identificador (por ejemplo, dirección de correo electrónico o nombre de usuario) y una contraseña. Esta funcionalidad la proporciona el FormLoginAuthenticator integrado.

En primer lugar vamos a crear un controlador (LoginController) para el formulario de inicio de sesión:

```
php bin/console make:controller Login
```

```php
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class LoginController extends AbstractController
{
    #[Route('/login', name: 'app_login')]
    public function index(): Response
    {
        return $this->render('login/index.html.twig', [
            'controller_name' => 'LoginController',
        ]);
    }
}
```

IMPORTANTE : A continuación, **habilita la autenticación con formulario de login con la configuración form_login del archivo** security.yaml

```yaml
# config/packages/security.yaml
security:
  # ...

  firewalls:
    main:
      # ...
      form_login:
        # "app_login" is the name of the route created previously
        login_path: app_login
        check_path: app_login
```

El **login_path y check_path** admititen URL´s y nombres de ruta (pero no pueden tener comodines, por ejemplo, /login/{foo}

Una vez habilitado, el sistema de seguridad **redirige a los visitantes no autenticados al login_path** cuando intentan acceder a un lugar seguro (este comportamiento se puede personalizar mediante puntos de entrada de autenticación).

Edita el controlador de inicio de sesión (LoginController) para representar el formulario de inicio de sesión:

```
// ...
+ use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;

 class LoginController extends AbstractController
```

```
 {
     #[Route('/login', name: 'app_login')]
-    public function index(): Response
+    public function index(AuthenticationUtils $authenticationUtils): Response
     {
+        // get the login error if there is one
+        $error = $authenticationUtils->getLastAuthenticationError();
+
+        // last username entered by the user
+        $lastUsername = $authenticationUtils->getLastUsername();
+
         return $this->render('login/index.html.twig', [
-            'controller_name' => 'LoginController',
+            'last_username' => $lastUsername,
+            'error'         => $error,
         ]);
     }
 }
```

No dejes que este controlador te confunda. Su trabajo es solo *representar* (render) el formulario. El componente `FormLoginAuthenticator` **controlará el envío del formulario automáticamente**. Si el usuario envía un correo electrónico o una contraseña no válidos, ese autenticador almacenará el error y lo redirigirá a este controlador, donde leemos el error (mediante `AuthenticationUtils`) para que se pueda mostrar al usuario.

Por último, cree o actualice la plantilla:

```twig
{# templates/login/index.html.twig #}
{% extends 'base.html.twig' %}

{# ... #}

{% block body %}
  {% if error %}
    <div>{{ error.messageKey|trans(error.messageData, 'security') }}</div>
  {% endif %}

  <form action="{{ path('app_login') }}" method="post">
    <label for="username">Email:</label>
    <input type="text" id="username" name="_username" value="{{ last_username }}">

    <label for="password">Password:</label>
    <input type="password" id="password" name="_password">

    {# If you want to control the URL the user is redirected to on success
    <input type="hidden" name="_target_path" value="/account"> #}

    <button type="submit">login</button>
  </form>
{% endblock %}
```

La variable de error que se pasa a la plantilla es una instancia de AuthenticationException. Puede contener información confidencial sobre el error de autenticación. *Nunca* uses error.message: use la propiedad messageKey en su lugar, como se muestra en el ejemplo. Este mensaje siempre es seguro de mostrar.

El formulario puede parecerse a cualquier cosa, pero por lo general sigue algunas convenciones:

1. El elemento <form> envía una solicitud POST a la ruta app_login, ya que eso es lo que configuró como check_path en la clave form_login en security.yaml;
2. El campo de nombre de usuario (o cualquiera que sea el "identificador" de su usuario, como un correo electrónico) tiene el nombre _username y el campo de contraseña tiene el nombre _password.

¡Y eso es todo! Al enviar el formulario, el sistema de seguridad lee automáticamente el _username y _password parámetro POST, carga al usuario a través del proveedor de usuarios, comprueba las credenciales del usuario y lo autentica o lo envía de vuelta al formulario de inicio de sesión donde se puede mostrar el error.

Para revisar todo el proceso:

1. El usuario intenta acceder a un recurso que está protegido (por ejemplo, /admin);
2. El firewall inicia el proceso de autenticación redirigiendo al usuario al formulario de inicio de sesión (/login);
3. La página /login representa el formulario de inicio de sesión a través de la ruta y el controlador creados en este ejemplo;
4. El usuario envía el formulario de inicio de sesión a /login;
5. El sistema de seguridad (es decir, FormLoginAuthenticator) intercepta la solicitud, comprueba las credenciales enviadas por el usuario, autentica al usuario si son correctas y envía al usuario de vuelta al formulario de inicio de sesión si no lo son.

7. Control de acceso (autorización)

Los usuarios pueden iniciar sesión en la aplicación mediante el formulario de inicio de sesión. Ahora, debes aprender a denegar el acceso y trabajar con el

objeto Usuario. A esto se le llama **autorización,** y su trabajo es decidir si un usuario puede acceder a algún recurso (una URL, un objeto del modelo, o una llamada a un método, ...).

El proceso de autorización tiene dos vertientes diferenciadas:

1. **El usuario recibe un rol específico al iniciar sesión** (por ejemplo, ROLE_ADMIN).
2. Agrega código para que un recurso (por ejemplo, URL, controlador,etc) requiera un "atributo" específico (por ejemplo, un rol como ROLE_ADMIN) para poder acceder a él.

8. Roles

Cuando un usuario inicia sesión, Symfony llama al método getRoles() en tu objeto Usuario para determinar qué roles tiene este usuario. En la clase User que se generó anteriormente, los roles son una matriz que se almacena en la base de datos y **a cada usuario *siempre* se le asigna al menos un rol: ROLE_USER:**

```php
#[ORM\Column]
private array $roles = [];
```

Puedes hacer *lo que* quieras para determinar qué roles debe tener un usuario. La única regla es que cada rol **debe comenzar con el prefijo ROLE_**, de lo contrario, las cosas no funcionarán. Aparte de eso, un rol es solo una cadena (String) y puedes crear lo que necesites (por ejemplo, ROLE_PRODUCT_ADMIN).

A continuación, usará estos roles para conceder acceso a secciones específicas de su sitio.

Roles jerárquicos

En lugar de asignar muchos roles a cada usuario, puede definir reglas de herencia de roles mediante la creación de una jerarquía de roles:

```yaml
# config/packages/security.yaml
security:
    # ...

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

Los usuarios con el rol ROLE_ADMIN también tendrán el rol ROLE_USER. Los usuarios con ROLE_SUPER_ADMIN, automáticamente tendrán ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH y ROLE_USER (heredados de ROLE_ADMIN).

Para que la jerarquía de roles funcione, no uses $user->getRoles() manualmente. Por ejemplo, en un controlador que se extiende desde el controlador base:

```
// BAD - $user->getRoles() will not know about the role hierarchy
$hasAccess = in_array('ROLE_ADMIN', $user->getRoles());

// GOOD - use of the normal security methods
$hasAccess = $this->isGranted('ROLE_ADMIN');
$this->denyAccessUnlessGranted('ROLE_ADMIN');
```

Agregar código para denegar el acceso

Hay **dos** formas de denegar el acceso a algo:

1. access_control en security.yaml te permite proteger determinadas patrones de URL (por ejemplo, /admin/*). Más simple, pero menos flexible;
2. en el controlador (controller) (u otro código).

Protección de patrones de URL (access_control)

La forma más básica de proteger parte de la aplicación es proteger un patrón de URL completo en security.yaml. Por ejemplo, para requerir ROLE_ADMIN para todas las direcciones URL que comienzan con /admin, puede:

```
# config/packages/security.yaml
security:
    # ...

    firewalls:
        # ...
        main:
            # ...

    access_control:
        # require ROLE_ADMIN for /admin*
        - { path: '^/admin', roles: ROLE_ADMIN }

        # or require ROLE_ADMIN or IS_AUTHENTICATED_FULLY for /admin*
        - { path: '^/admin', roles: [IS_AUTHENTICATED_FULLY, ROLE_ADMIN] }

        # the 'path' value can be any valid regular expression
        # (this one will match URLs like /api/post/7298 and /api/comment/528491)
```

```
    - { path: ^/api/(post|comment)/\d+$, roles: ROLE_USER }
```

Puedes definir tantos patrones de URL como necesites, cada uno es una expresión regular. **Pero**, solo **se** emparejará uno por solicitud: Symfony comienza en la parte superior de la lista y se detiene cuando encuentra la primera coincidencia:

```
# config/packages/security.yaml
security:
  # ...

  access_control:
    # matches /admin/users/*
    - { path: '^/admin/users', roles: ROLE_SUPER_ADMIN }

    # matches /admin/* except for anything matching the above rule
    - { path: '^/admin', roles: ROLE_ADMIN }
```

Anteponer la ruta de acceso con ^ significa que solo coinciden las direcciones URL que *comienzan* con el patrón. Por ejemplo, una ruta de /admin (sin el ^) coincidiría con /admin/foo, pero también coincidiría con URL como /foo/admin.

Cada access_control también puede coincidir con la dirección IP, el nombre de host y los métodos HTTP. También se puede utilizar para redirigir a un usuario a la versión https de un patrón de URL. Para necesidades más complejas, también puede usar un servicio que implemente RequestMatcherInterface.

Protección de controladores y otros códigos

Puede denegar el acceso desde el interior de un controlador:

```
// src/Controller/AdminController.php
// ...

public function adminDashboard(): Response
{
  $this->denyAccessUnlessGranted('ROLE_ADMIN');

  // or add an optional message - seen by developers
  $this->denyAccessUnlessGranted('ROLE_ADMIN', null, 'User tried to access a page without
having ROLE_ADMIN');
}
```

¡Eso es todo! Si no se concede acceso, se produce una excepción AccessDeniedException especial y no se llama a más código en el controlador. Entonces, sucederá una de estas dos cosas:

1. Si el usuario aún no ha iniciado sesión, se le pedirá que inicie sesión (por ejemplo, se le redirigirá a la página de inicio de sesión).

2. Si el usuario *ha* iniciado sesión, pero no tiene el rol ROLE_ADMIN, se le mostrará la página 403 de acceso denegado (que puede personalizar).

Otra forma de proteger una o varias acciones del controlador es usar el atributo #[IsGranted()]. En el siguiente ejemplo, todas las acciones del controlador requerirán el permiso ROLE_ADMIN, excepto adminDashboard(), que requerirá el permiso ROLE_SUPER_ADMIN:

```
// src/Controller/AdminController.php
// ...

use Symfony\Component\Security\Http\Attribute\IsGranted;

#[IsGranted('ROLE_ADMIN')]
class AdminController extends AbstractController
{
   // Optionally, you can set a custom message that will be displayed to the user
   #[IsGranted('ROLE_SUPER_ADMIN', message: 'You are not allowed to access the admin dashboard.')]
   public function adminDashboard(): Response
   {
      // ...
   }
}
```

Permitir el acceso no seguro (usuarios anónimos)

Cuando un visitante aún no ha iniciado sesión en su sitio web, se le trata como "no autenticado" y no tiene ningún rol. Esto les impedirá visitar tus páginas si definiste una regla access_control.

En la configuración access_control, puede utilizar el atributo de seguridad PUBLIC_ACCESS para excluir algunas rutas para el acceso no autenticado (por ejemplo, la página de inicio de sesión):

```
# config/packages/security.yaml
security:

   # ...
   access_control:
      # allow unauthenticated users to access the login form
      - { path: ^/admin/login, roles: PUBLIC_ACCESS }

      # but require authentication for all other admin routes
      - { path: ^/admin, roles: ROLE_ADMIN }
```

Comprobación de si un usuario ha iniciado sesión

Si *solo* deseas verificar si un usuario ha iniciado sesión (sin importar los roles), tienes las siguientes dos opciones.

1. En primer lugar, si has dado *ROLE_USER* a todos los usuarios, puedes comprobar ese rol.
2. En segundo lugar, puede usar el IS_AUTHENTICATED_FULLY especial "atributo" en lugar de un rol:

```
// ...

public function adminDashboard(): Response
{
   $this->denyAccessUnlessGranted('IS_AUTHENTICATED');

   // ...
}
```

Comprender cómo se actualizan los usuarios desde la sesión

Al final de cada solicitud (a menos que el firewall no tenga estado), **el objeto User se serializa en la sesión**. Al comienzo de la siguiente solicitud, se deserializa y luego se pasa a su proveedor de usuarios para que la "actualice" (por ejemplo, consultas de Doctrine para un usuario nuevo).

Luego, los dos objetos User (el original de la sesión y el objeto User actualizado) se "comparan" para ver si son "iguales". De forma predeterminada, la clase principal AbstractToken compara los valores devueltos de los métodos getPassword(), getSalt() y getUserIdentifier(). Si alguno de estos es diferente, se cerrará la sesión del usuario. Se trata de una medida de seguridad para garantizar que los usuarios malintencionados puedan ser desautenticados si los datos principales de los usuarios cambian.

Sin embargo, en algunos casos, este proceso puede causar problemas de autenticación inesperados. Si tiene problemas para autenticarse, es posible que *se* esté autenticando correctamente, pero pierda inmediatamente la autenticación después de la primera redirección.

En ese caso, revise la lógica de serialización (por ejemplo, los métodos __serialize() o serialize()) en su clase de usuario (si tiene alguna) para asegurarse de que todos los campos necesarios estén serializados y también excluya todos los campos que no es necesario serializar (por ejemplo, relaciones de doctrine).

Redireccionamiento después del éxito en el Login

De forma predeterminada, el formulario redirigirá a la URL solicitada por el usuario (es decir, la URL que activó el formulario de inicio de sesión que se muestra). Por ejemplo, si el usuario solicitó http://www.example.com/admin/post/18/edit, una vez que haya iniciado sesión correctamente, se le enviará de vuelta a http://www.example.com/admin/post/18/edit.

Esto se hace almacenando la URL solicitada en la sesión. Si no hay ninguna URL presente en la sesión (tal vez el usuario fue directamente a la página de inicio de sesión), entonces el usuario es redirigido a `/` (es decir, la página de inicio). Puede cambiar este comportamiento de varias maneras.

Cambiar la página predeterminada a la que se va cuando se hace login

Define la opción `default_target_path` para cambiar la página a la que se redirige al usuario **si no se almacenó ninguna página anterior en la sesión**. El valor puede ser una URL relativa/absoluta o un nombre de ruta de Symfony:

```yaml
# config/packages/security.yaml
security:
  # ...

  firewalls:
    main:
      form_login:
        # ...
        default_target_path: after_login_route_name
```

Redirigir siempre a la página predeterminada

Define la opción booleana **always_use_default_target_path** para ignorar la URL solicitada anteriormente y redirigir siempre a la página predeterminada:

```yaml
# config/packages/security.yaml
security:
  # ...

  firewalls:
    main:
      form_login:
        # ...
        always_use_default_target_path: true
```

Redireccionamiento después de un error de Login

Después de un inicio de sesión fallido (por ejemplo, se envió un nombre de usuario o contraseña no válidos), el usuario es redirigido de nuevo al formulario de inicio de sesión. Utiliza la opción failure_path para definir un nuevo destino a través de una URL relativa/absoluta o un nombre de ruta de Symfony:

```yaml
# config/packages/security.yaml
security:
  # ...

  firewalls:
    main:
      # ...
      form_login:
        # ...
        failure_path: login_failure_route_name
```