Unidad 9 - Seguridad

La seguridad es un proceso de dos etapas, cuyo objetivo es evitar que un usuario acceda a un recurso al cual no debería tener acceso.

En el primer paso del proceso, el sistema de seguridad identifica quién es el usuario obligándolo a presentar algún tipo de identificación. Esto se llama autenticación, y significa que el sistema está tratando de determinar quién eres.

Una vez que el sistema sabe quien eres, el siguiente paso es determinar si deberías tener acceso a un determinado recurso. Esta parte del proceso se llama autorización, y significa que el sistema está comprobando si tienes suficientes privilegios para realizar una determinada acción.

El usuario

Los permisos en Symfony siempre están vinculados a un objeto de usuario. Si necesitamos proteger (partes de) la aplicación, debemos crear una clase Usuario. Esta es una clase que implementa UserInterface. Esta suele ser una entidad de Doctrine, pero también puede usar una clase de usuario de seguridad personalizada.

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

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

 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
```

Este comando crea una entidad Usuario con el siguiente formato:

```php
// src/Entity/User.php
namespace App\Entity;

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

/**
 * @ORM\Entity(repositoryClass=UserRepository::class)
 */
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=180, unique=true)
     */
    private $email;

    /**
     * @ORM\Column(type="json")
     */
    private $roles = [];

    /**
     * @var string The hashed password
     * @ORM\Column(type="string")
     */
    private $password;

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

    public function getEmail(): ?string
    {
        return $this->email;
    }

    public function setEmail(string $email): self
    {
        $this->email = $email;

        return $this;
    }
```

```
    /**
      * The public representation of the user (e.g. a username, an email address,
etc.)
     *
     * @see UserInterface
     */
    public function getUserIdentifier(): string
    {
        return (string) $this->email;
    }

    /**
     * @deprecated since Symfony 5.3
     */
    public function getUsername(): string
    {
        return (string) $this->email;
    }

    /**
     * @see UserInterface
     */
    public function getRoles(): array
    {
        $roles = $this->roles;
        // guarantee every user at least has ROLE_USER
        $roles[] = 'ROLE_USER';

        return array_unique($roles);
    }

    public function setRoles(array $roles): self
    {
        $this->roles = $roles;

        return $this;
    }

    /**
     * @see PasswordAuthenticatedUserInterface
     */
    public function getPassword(): string
    {
        return $this->password;
    }

    public function setPassword(string $password): self
    {
        $this->password = $password;

        return $this;
    }
```

```
    /**
     * Returning a salt is only needed, if you are not using a modern
     * hashing algorithm (e.g. bcrypt or sodium) in your security.yaml.
     *
     * @see UserInterface
     */
    public function getSalt(): ?string
    {
        return null;
    }

    /**
     * @see UserInterface
     */
    public function eraseCredentials()
    {
        // If you store any temporary, sensitive data on the user, clear it here
        // $this->plainPassword = null;
    }
}
```

Luego debemos crear las tablas en la base de datos a través de los comandos de migración:

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

Además de crear la entidad, el comando make:user también agrega configuración para un proveedor de usuario en su configuración de seguridad:

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

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

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

Los proveedores de usuarios se utilizan en un par de lugares durante el ciclo de vida de la seguridad:

- **Cargar el Usuario en base a un identificador:** Durante el inicio de sesión (o cualquier otro autenticador), el proveedor carga al usuario en función del identificador de usuario.
- **Recargar el Usuario desde la sesión:** Al comienzo de cada solicitud, el usuario se carga desde la sesión. El proveedor "actualiza" 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 (y, si es necesario, se anula la autenticación o se cierra la sesión del usuario si algo cambia).

Generación de contraseñas

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

Hay que asegurarse que la clase Usuario implemente PasswordAuthenticatedUserInterface.

```
// src/Entity/User.php

// ...
use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;

class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    // ...

    /**
     * @return string the hashed password for this user
     */
    public function getPassword(): string
    {
        return $this->password;
    }
}
```

Luego, debermos configurar qué hasher de contraseña debe usarse para esta clase. Si el archivo security.yaml aún no estaba preconfigurado, entonces make:user debería haber hecho esto:

```
# config/packages/security.yaml
security:
    # ...
    password_hashers:
        Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface:
'auto'
```

Ahora que Symfony sabe cómo queremos codificar las contraseñas, se puede usar el servicio UserPasswordHasherInterface para codificar la contraseña antes de guardar a los usuarios en la base de datos:

```
// src/Controller/RegistrationController.php
namespace App\Controller;

// ...
use Symfony\Component\PasswordHasher\Hasher\UserPasswordHasherInterface;

class RegistrationController extends AbstractController
{
    public function index(UserPasswordHasherInterface $passwordHasher)
    {
        // ... e.g. get the user data from a registration form
        $user = new User(...);
        $plaintextPassword = ...;

        // hash the password (based on the security.yaml config)
        $hashedPassword = $passwordHasher->hashPassword(
            $user,
            $plaintextPassword
        );
        $user->setPassword($hashedPassword);

        // ...
    }
}
```

También podemos codificar manualmente una contraseña ejecutando:

```
php bin/console security:hash-password
```

Autenticación

Cortafuegos

La sección de firewalls de config/packages/security.yaml es la sección más importante. Un "cortafuegos" es su sistema de autenticación: el cortafuegos define qué partes de la aplicación están protegidas y cómo sus usuarios podrán autenticarse.

```
# config/packages/security.yaml
security:
    # ...
    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            lazy: true
            provider: users_in_memory
```

Solo un cortafuegos está activo en cada solicitud: Symfony usa la clave pattern para encontrar la primera coincidencia.

El cortafuegos de desarrollo es en realidad un cortafuegos falso: se asegura de que no bloquees accidentalmente las herramientas de desarrollo de Symfony, que se encuentran en direcciones URL como /_profiler y /_wdt.

Todas las URL reales son manejadas por el firewall principal (no definir la clave pattern significa que coincide con todas las URL). Un firewall puede tener muchos modos de autenticación, en otras palabras, permite muchas formas de hacer la pregunta "¿Quién eres?".

Durante la autenticación, el sistema intenta encontrar un usuario que coincida con el visitante de la página web. Tradicionalmente, esto se hacía mediante un formulario de inicio de sesión o un cuadro de diálogo básico HTTP en el navegador. Sin embargo, Symfony viene con muchos otros autenticadores:

- Form Login
- JSON Login
- HTTP Basic
- Login Link
- X.509 Client Certificates
- Remote users
- Custom Authenticators

Form 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 autenticador de inicio de sesión de formulario.

Veamos cómo hacerlo. Primero habilitamos el autenticador de inicio de sesión de formulario usando la configuración form_login en el security.yaml

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

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

Una vez habilitado, el sistema de seguridad redirige a los visitantes no autenticados a login_path cuando intentan acceder a un lugar seguro.

Luego definimos el controlador:

```
// src/Controller/LoginController.php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;

class LoginController extends AbstractController
{
    /**
     * @Route("/login", name="app_login")
     */
    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', [
             'last_username' => $lastUsername,
             'error'         => $error,
          ]);
    }
}
```

Este controlador no hace nada con respecto a la autenticación. Su trabajo es solo presentar el formulario: el autenticador form_login manejará 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.

Finalmente creamos el template

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

{# ... #}

{% block body %}
    {% if error %}
        <div>Usuario o contraseña incorrecta</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"/>

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

Para revisar todo el proceso:

- El usuario intenta acceder a un recurso que está protegido (por ejemplo, /admin);
- El cortafuegos inicia el proceso de autenticación redirigiendo al usuario al formulario de inicio de sesión (/login);
- La página /login presenta el formulario de inicio de sesión a través de la ruta y el controlador creados;
- El usuario envía el formulario de inicio de sesión a /login;
- El sistema de seguridad (es decir, el autenticador form_login) intercepta la solicitud, verifica las credenciales enviadas por el usuario, autentica al usuario si son correctas y lo devuelve al formulario de inicio de sesión si no lo son.

Logout

Para habilitar el cierre de sesión, se debe activar el parámetro de configuración de cierre de sesión en el firewall:

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

    firewalls:
        main:
            # ...
            logout:
                path: app_logout
```

Creamos una ruta en el controlador que no haga nada:

```
    /**
     * @Route("/logout", name="app_logout", methods={"GET"})
     */
    public function logout(): void
    {
        // controller can be blank: it will never be called!
       throw new \Exception('Don\'t forget to activate logout in security.yaml');
    }
```

Acceder al objeto usuario

Después de la autenticación, se puede acceder al objeto Usuario del usuario actual a través del método getUser() en el controlador:

```
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class ProfileController extends AbstractController
{
    public function index(): Response
    {
        // returns your User object, or null if the user is not authenticated
        $user = $this->getUser();

        // Call whatever methods you've added to your User class
        // For example, if you added a getFirstName() method, you can use that.
        return new Response('Well hi there '.$user->getFirstName());
    }
}
```

En una plantilla Twig, el objeto de usuario está disponible a través de la variable app.user:

```
{% if is_granted('IS_AUTHENTICATED_FULLY') %}
    <p>Email: {{ app.user.email }}</p>
{% endif %}
```

Control de acceso

Los usuarios ahora pueden iniciar sesión en su aplicación usando su formulario de inicio de sesión. Ahora, debemos autorizar el acceso y trabajar con el objeto Usuario. Esto se llama autorización, y su trabajo es decidir si un usuario puede acceder a algún recurso (una URL, un objeto modelo, una llamada a un método, etc.).

Roles

Cuando un usuario inicia sesión, Symfony llama al método getRoles() en tu objeto Usuario para determinar qué roles tiene este usuario. Este método debe devolver un array de strings con cada uno de los roles que posee el usuario.

```
// src/Entity/User.php

// ...
class User
{
    /**
     * @ORM\Column(type="json")
     */
    private $roles = [];

    // ...
    public function getRoles(): array
    {
        $roles = $this->roles;
        // guarantee every user at least has ROLE_USER
        $roles[] = 'ROLE_USER';

        return array_unique($roles);
    }
}
```

Los nombres de los roles deben seguir algunas pautas:

- Cada rol debe comenzar con ROLE_
- Aparte de la regla anterior, un rol es solo una cadena y puede inventar lo que necesita (por ejemplo, ROLE_PRODUCT_ADMIN).

Estos roles se utilizarán a continuación para otorgar acceso a secciones específicas del sitio.

Agregar código para denegar el acceso

Hay dos formas de denegar el acceso a algo:

- **access_control en security.yaml:** te permite proteger los patrones de URL (por ejemplo, /admin/*). Más simple, pero menos flexible;
- **en su controlador** (u otro código).

Securizando con patrones de URL

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 URL que comienzan con /admin, sería de la siguiente manera:

```
# 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 }
```

Puede definir tantos patrones de URL como necesite; cada uno es una expresión regular. Pero, solo una coincidirá 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 ^ a la ruta significa que solo se emparejan las URL que comienzan con el patrón. Por ejemplo, una ruta de /admin (sin ^) coincidiría con /admin/foo pero también coincidiría con direcciones URL como /foo/admin.

Securizando Controllers

Se 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');
}
```

También se pueden utilizar anotaciones:

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

use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;

/**
 * Require ROLE_ADMIN for all the actions of this controller
 *
 * @IsGranted("ROLE_ADMIN")
 */
class AdminController extends AbstractController
{
    /**
     * Require ROLE_SUPER_ADMIN only for this action
     *
     * @IsGranted("ROLE_SUPER_ADMIN")
     */
    public function adminDashboard(): Response
    {
        // ...
    }
}
```

Control de accesos en templates

Si se desea verificar si el usuario actual tiene un rol determinado, se puede usar la función de ayuda integrada is_granted() en cualquier plantilla de Twig:

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

Permitir áreas inseguras

Cuando un visitante aún no ha iniciado sesión en su sitio web, se lo trata como "no autenticado" y no tiene ningún rol. Esto impedirá que visiten sus páginas si definió una regla de control de acceso.

En la configuración de access_control, puede usar 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:
    enable_authenticator_manager: true

    # ...
    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 }
```