



# Desarrollo de aplicaciones con Symfony 4

Security



# Security: Instalación

En la aplicaciones que dispongan de symfony flex y que no hayan sido instaladas con **symfony/website-skeleton**, donde este componente viene por defecto, ejecutaremos:

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

# Security: Creando la clase user

Creamos nuestra clase User con el maker:

```
$ php bin/console make:user
```

Y tras una serie de preguntas veremos:

```
created: src/Entity/User.php
```

```
created: src/Repository/UserRepository.php
```

```
updated: src/Entity/User.php
```

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

# Security: Creando la clase user

- La única regla sobre su clase de usuario es que debe implementar `ManagerInterface`.
- Podemos agregar cualquier otro campo o lógica que necesite.
- Podemos usar el comando `make: entity` para agregar más campos.

# Security: Explicando el security.yaml

## User Provider

Además de su clase de Usuario, también necesita un "Proveedor de Usuario": una clase que ayuda con algunas cosas, como volver a cargar los datos del Usuario de la sesión y algunas características opcionales, como recordar y la personificación.

providers:

app\_user\_provider:

entity:

class: App\Entity\User

property: email

# Security: Explicando el security.yaml

Como nuestra clase de usuario es una entidad no necesitamos hacer ninguna configuración adicional.

Por el contrario, si la clase de usuario no fuese una entidad, make: user también habrá generado una clase UserProvider que necesita terminar de configurar. Para más información:

[https://symfony.com/doc/current/security/user\\_provider.html](https://symfony.com/doc/current/security/user_provider.html)

# Security: Explicando el security.yaml

## Codificando contraseñas

En el apartado encoders le vamos a decir el algoritmo que vamos a utilizar para codificar la contraseña:

encoders:

App\Entity\User:

algorithm: argon2i

Se recomiendan bcrypt o argon2i

argon2i es más seguro, pero requiere PHP 7.2 o la extensión Sodium

# Security: Explicando el security.yaml

Como Symfony sabe cómo desea codificar las contraseñas, podemos usar el servicio `UserPasswordEncoderInterface` para hacer esto antes de guardar sus usuarios en la base de datos.

```
public function __construct(UserPasswordEncoderInterface $passwordEncoder) {  
    $this->passwordEncoder = $passwordEncoder;  
}  
  
public function create($user) {  
    $user = new User();  
    $user->setPassword( $this->passwordEncoder->encodePassword(  
        $user,  
        'the_new_password'  
    ) );  
}
```



# Security: Explicando el security.yaml

Para codificar manualmente las contraseñas tenemos el command

```
$ php bin/console security:encode-password
```

# Security: Explicando el security.yaml

## Firewalls

firewalls:

dev:

pattern: ^/(\_(profiler|wdt)|css|images|js)/

security: false

main:

pattern: ^/admin(\$|(/.\*))\$

provider: app\_user\_provider

anonymous: ~

guard:

authenticators:

- App\Security\LoginFormAuthenticator

# <https://symfony.com/doc/current/reference/configuration/security.html>

# Security: Crear el formulario de inicio de sesión

```
$ php bin/console make:auth
```

What style of authentication do you want? [Empty authenticator]:

[0] Empty authenticator

[1] Login form authenticator

> 1

The class name of the authenticator to create (e.g. AppCustomAuthenticator):

> LoginFormAuthenticator

Choose a name for the controller class (e.g. SecurityController) [SecurityController]:

> SecurityController

# Security: Crear el formulario de inicio de sesión

Deberemos ver el siguiente resultado:

created: src/Security/LoginFormAuthenticator.php

updated: config/packages/security.yaml

created: src/Controller/SecurityController.php

created: templates/security/login.html.twig

# Security: Crear el formulario de inicio de sesión

Deberemos ver el siguiente resultado:

created: src/Security/LoginFormAuthenticator.php

updated: config/packages/security.yaml

created: src/Controller/SecurityController.php

created: templates/security/login.html.twig

# Security: Crear el formulario de inicio de sesión

Editamos el archivo security.yml para permitir el acceso de cualquier persona a la ruta / login:

```
# config/packages/security.yml
security:
    # ...
    access_control:
        - { path: ^/login$, roles: IS_AUTHENTICATED_ANONYMOUSLY }
    # ...
```

# Security: Crear el formulario de inicio de sesión

Actualizamos el security.yaml para habilitar el autenticador Guard:

```
# config/packages/security.yaml
security:
    # ...
    firewalls:
        main:
            # ...
            guard:
                authenticators:
                    - App\Security\LoginFormAuthenticator
```

# Security: Crear el formulario de inicio de sesión

El comando `make:auth` acaba de hacer mucho trabajo por nosotros. Pero todavía quedan cosas por hacer.

Cuando enviamos el formulario, el `LoginFormAuthenticator` interceptará la solicitud, leerá el correo electrónico (o el campo que esté usando) y la contraseña del formulario, buscará el objeto `Usuario`, validará el token CSRF y verificará la contraseña, pero, dependiendo de su configuración, deberemos terminar una o más tareas pendientes antes de que funcione todo el proceso. Al menos deberemos completar donde deseamos que se redirija a su usuario después del éxito:



# Security: Crear el formulario de inicio de sesión

```
// src/Security/LoginFormAuthenticator.php
```

```
// ...
```

```
public function onAuthenticationSuccess(Request $request, TokenInterface $token,  
$providerKey)
```

```
{
```

```
    // ...
```

```
-    throw new \Exception('TODO: provide a valid redirect inside '.__FILE__);
```

```
+    // redirect to some "app_homepage" route - of wherever you want
```

```
    return new RedirectResponse($this->urlGenerator->generate('app_homepage'));
```

# Security: Crear el formulario de inicio de sesión

Ahora ya podemos logearnos en nuestra aplicación (lógicamente, habrá que crear algún usuario).

Podremos comprobar que un login ha sido correcto revisando la barra de herramientas de depuración web, que nos dirá quién somos y qué roles tenemos:

```
Logged in as  user1@symfony.com
Authenticated Yes
Token class   PostAuthenticationGuardToken
Firewall name main
```

200

@ app\_homepage

168 ms

2.0 MB



10



user1@symfony.com



6 ms



1

# Security: Seguridad en el controller

Podemos utilizar anotaciones:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGranted;
```

```
/**
```

```
 * Require ROLE_ADMIN for only this controller method.
```

```
 *
```

```
 * @IsGranted("ROLE_ADMIN")
```

```
 */
```

# Security: Seguridad en el controller

O podemos denegar el acceso desde dentro de un controlador:

```
public function adminDashboard()  
{  
    $this->denyAccessUnlessGranted('ROLE_ADMIN');  
  
    // añadir un mensaje opcional  
    $this->denyAccessUnlessGranted('ROLE_ADMIN', null, '¡Donde vas!');  
}
```

# Security: Seguridad en las vistas.

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

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

# Security: Seguridad en los servicios.

```
use Symfony\Component\Security\Core\Security;
//...
protected $security;
//...
public function __construct(Security $security) {
    $this->security = $security;
}
//...
if ($this->security->isGranted('ROLE_SALES_ADMIN')) { ... }
```

Ejemplo completo:

[https://symfony.com/doc/current/security/securing\\_services.html](https://symfony.com/doc/current/security/securing_services.html)

# Security

## Comprobando si un usuario ha iniciado sesión (IS\_AUTHENTICATED\_FULLY)

Si solo deseamos verificar si un usuario está conectado (no nos importan los roles), tenemos dos opciones. Primero, si le ha otorgado a cada usuario ROLE\_USER, solo puede verificar ese rol. De lo contrario, puede usar un "atributo" especial en lugar de un rol:

```
$this->denyAccessUnlessGranted('IS_AUTHENTICATED_FULLY');
```

# Security

Podemos usar `IS_AUTHENTICATED_FULLY` en cualquier lugar que se utilicen los roles: como `access_control` o en Twig.

`IS_AUTHENTICATED_FULLY` no es un rol, pero actúa como uno, y todos los usuarios que han iniciado sesión tendrán esto. En realidad, hay 3 atributos especiales como este:

`IS_AUTHENTICATED_REMEMBERED`: Todos los usuarios registrados tienen esto, incluso si han iniciado sesión debido al uso "remember me cookie".



# Security

IS\_AUTHENTICATED\_FULLY: Es similar a IS\_AUTHENTICATED\_REMEMBERED, pero más fuerte. Los usuarios que hayan iniciado sesión solo debido a una "remember me cookie" tendrán IS\_AUTHENTICATED\_REMEMBERED pero no tendrán IS\_AUTHENTICATED\_FULLY.

IS\_AUTHENTICATED\_ANONYMOUSLY: Todos los usuarios (incluso los anónimos) tienen esto; esto es útil cuando se incluyen listas de direcciones URL para garantizar el acceso. Podéis leer más acerca de esto:

[https://symfony.com/doc/current/security/access\\_control.html](https://symfony.com/doc/current/security/access_control.html)

# Security: Obtener el objeto usuario

En el controlador:

```
$user = $this->getUser();
```

En un servicio (necesitaremos inyectar el servicio Security):

```
use Symfony\Component\Security\Core\Security;
```

```
//...
```

```
public function __construct(Security $security)
```

```
// ...
```

```
$user = $this->security->getUser();
```

# Security: Obtener el objeto usuario

En una vista:

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

# Security: Logging Out

Para habilitar el cierre de sesión, deberemos activar el parámetro de configuración de cierre de sesión en su firewall:

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

```
security:
```

```
    # ...
```

```
    firewalls:
```

```
        main:
```

```
            # ...
```

```
            logout:
```

```
                path: app_logout
```

```
                target: app_any_route # ruta a la que redireccionamos tras el logout
```

# Security: Logging Out

A continuación, debermos crear una ruta para esta URL (pero no un controlador):

```
# config/routes.yaml  
app_logout:  
    path: /logout  
    methods: GET
```

# Security: Roles jerárquicos

En lugar de otorgar muchos roles a cada usuario, podemos definir reglas de herencia de roles creando una jerarquía de roles:

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

```
security:
```

```
  # ...
```

```
  role_hierarchy:
```

```
    ROLE_ADMIN:    ROLE_USER
```

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

# Security: Roles jerárquicos

Los usuarios con el rol `ROLE_ADMIN` también tendrán el rol `ROLE_USER`. Y los usuarios con `ROLE_SUPER_ADMIN`, automáticamente tendrán `ROLE_ADMIN`, `ROLE_ALLOWED_TO_SWITCH` y `ROLE_USER` (heredado de `ROLE_ADMIN`).

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

```
$hasAccess = in_array('ROLE_ADMIN', $user->getRoles());
```

```
$hasAccess = $this->isGranted('ROLE_ADMIN');
```

```
$this->denyAccessUnlessGranted('ROLE_ADMIN');
```

# Security: ¡Importante!

La seguridad no funciona en las páginas de error.

Como el enrutamiento se realiza antes que la seguridad, las páginas de error 404 no están cubiertas por ningún firewall. Esto significa que no podemos verificar la seguridad ni acceder al objeto de usuario en estas páginas

Ver: [https://symfony.com/doc/current/controller/error\\_pages.html](https://symfony.com/doc/current/controller/error_pages.html) para más detalles.



# Security: Más sobre el security

¿Quieres profundizar un más en la seguridad o hacer algo diferente?

<https://symfony.com/doc/current/security.html#learn-more>

Security voters:

<https://symfony.com/doc/current/security/voters.html>

Cómo securizar una api:

[https://symfony.com/doc/current/security/guard\\_authentication.html](https://symfony.com/doc/current/security/guard_authentication.html)

¿Preguntas?

