UD-13: Seguridad y control de acceso

Desarrollo Web en Entorno Servidor

Curso 2020/2021

INTRODUCCIÓN

En esta unidad aprenderemos a establecer los principales elementos del sistema de seguridad de Symfony :

- El mecanismo de **autenticación**, es decir, establecer dónde están registrados los usuarios para poder acceder a ellos y validar las credenciales de acceso a la aplicación.
- El mecanismo de **autorización**, es decir, una vez se ha validado el usuario que accede, y siempre que éste sea correcto, determinar sus permisos y a qué recursos puede acceder y a cuáles no.

Empezaremos a trabajar con el archivo ***config/packages/security.yaml***, que es donde se almacena la configuración general del sistema de seguridad de nuestra aplicación Symfony.

ESTABLECER MODO AUTENTICACIÓN Y ORIGEN DE DATOS

Veamos cómo añadir elementos de autenticación en este apartado de configuración:

- Podemos establecer distintos modos de autenticación, y distintas fuentes de datos de donde tomar los usuarios para validar. El mecanismo menos recomendado es utilizar una autenticación básica de HTTP (la que muestra un "prompt" básico para introducir login y password), almacenando los usuarios en el propio archivo *security.yaml*. Nos saltaremos esta opción.
- La opción que realmente nos interesa es tener los usuarios registrados en una tabla de una base de datos MySQL. Definiremos un formulario de *login* que utilizaremos para validarnos contra a la hora de acceder a recursos protegidos.
  - En nuestra aplicación de contactos, vamos a añadir una nueva entidad para almacenar los usuarios de la aplicación, y su correspondiente tabla asociada, usando Doctrine.
  - Los usuarios tendrán por tanto un ***id*** (autogenerado por Doctrine), un ***login***, un ***password***, un ***email*** y un ***rol*** (más adelante veremos cómo utilizar el ***rol***).

DEFINIR LA ENTIDAD Y LA TABLA DE USUARIOS

```
php bin/console make:entity

Class name of the entity to create or update:
> Usuario
created: src/Entity/Usuario.php
created: src/Repository/UsuarioRepository.php
Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-
running this command.
New property name (press <return> to stop adding fields):
> login
Field type (enter ? to see all types) [string]:
> string
Field length [255]:
> 100
Can this field be null in the database (nullable)
(yes/no) [no]:
> no
updated: src/Entity/Usuario.php
Add another property? Enter the property name (or press
<return> to stop adding fields):
> password
Field type (enter ? to see all types) [string]:
> string
Field length [255]:
> 100
Can this field be null in the database (nullable)
(yes/no) [no]:
> no
updated: src/Entity/Usuario.php
Add another property? Enter the property name (or press
<return> to stop adding fields):
> email
Field type (enter ? to see all types) [string]:
> string
Field length [255]:
> 100
Can this field be null in the database (nullable) (yes/no)
[no]:
> no
updated: src/Entity/Usuario.php
Add another property? Enter the property name (or press
<return> to stop adding fields):
> rol
Field type (enter ? to see all types) [string]:
> string
Field length [255]:
> 20
Can this field be null in the database (nullable) (yes/no)
[no]:
> no
updated: src/Entity/Usuario.php
Add another property? Enter the property name (or press
<return> to stop adding fields):
>
Success!
```

DEFINIR LA ENTIDAD Y LA TABLA DE USUARIOS

- Una vez creada la entidad, el siguiente paso es migrar los cambios a la base de datos, como hemos hecho otras veces:

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

- Podemos añadir a mano un usuario de prueba en la BD, con estos atributos:

login	password	email	rol
pepe	1234	pepe@gmail.com	ROLE_USER

- Veremos los roles más adelante. De momento dejaremos establecido éste para este usuario de prueba.

IMPLEMENTAR LAS INTERFACES REQUERIDAS

Para poder usar una entidad como fuente de usuarios que loguear, es necesario que dicha entidad **implemente** la **interfaz** `UserInterface`, lo que obliga a definir algunos métodos adicionales:

- `getRoles()`, que devolverá un array con los roles del usuario (en nuestro caso, cada usuario sólo tendrá un rol, pero habrá que devolverlo en un array).
- `getPassword()`, que devolverá el password del usuario.
- `getUserName()`, que devolverá el login del usuario.
- `getSalt()`, que en ocasiones no es necesario emplear. Se usa en mecanismos de codificación de passwords más avanzados que los que veremos en esta asignatura. Así que en nuestro ejemplo devolveremos *null*.
- `eraseCredentials()`, que se usa para eliminar información privada del usuario. Puede ser útil si, por ejemplo, se almacena el password del usuario sin encriptar. Primero lo haremos así, pero luego lo encriptaremos, así que lo vamos a dejar vacío por ahora.

Además, conviene implementar la interfaz `Serializable` para poder serializar objetos de tipo Usuario y enviarlos entre las partes de la aplicación. Esto implica añadir dos métodos más: `serialize` (para convertir el usuario en texto que enviar entre componentes) y `unserialize` (para convertir un texto en un objeto **Usuario**).

IMPLEMENTAR LAS INTERFACES REQUERIDAS

```php
<?php

    namespace App\Entity;

    use Doctrine\ORM\Mapping as ORM;
    use Symfony\Component\Security\Core\User\UserInterface;

    /**
    * @ORM\Entity(repositoryClass="App\Repository\UsuarioRepository")
    */
    class Usuario implements UserInterface, \Serializable
    {
        /**
        * @ORM\Id
        * @ORM\GeneratedValue()
        * @ORM\Column(type="integer")
        */
        private $id;
        /**
        * @ORM\Column(type="string", length=100)
        */
        private $login;
        /**
        * @ORM\Column(type="string", length=100)
        */
        private $password;
```

Aquello añadido en morado lo hemos tenido que añadir a mano para implementar correctamente la interfaz **UserInterface**.

Implementamos esta interfaz para que nuestra entidad **Usuario** pueda funcionar dentro del sistema de seguridad de Symfony.

IMPLEMENTAR LAS INTERFACES REQUERIDAS

```
/**
* @ORM\Column(type="string", length=100)
*/
private $email;
/**
* @ORM\Column(type="string", length=20)
*/
private $rol;

public function getUserName()
{
     return $this->login;
}
public function getSalt()
{
     return null;
}
public function getRoles()
{
     return array($this->rol);
}
public function eraseCredentials()
{
}
```

Todos los métodos en morado los hemos tenido que añadir a mano para implementar correctamente la interfaz **UserInterface**.

Implementamos esta interfaz para que nuestra entidad **Usuario** pueda funcionar dentro del sistema de seguridad de Symfony.

IMPLEMENTAR LAS INTERFACES REQUERIDAS

```
        public function serialize()
        {
            return serialize(array($this->id, $this->login, $this->password));
        }

        public function unserialize($datos_serializados)
        {
            list($this->id, $this->login, $this->password) = unserialize($datos_serializados,
                                                              array('allowed_classes'=> false));
        }
        ...
    }
?>
```

Los métodos en azul también los hemos tenido que añadir a mano para implementar correctamente la interfaz **Serializable**.

Implementamos esta interfaz para poder enviar objetos de tipo **Usuario** de un sitio a otro.

A la función **unserialize**, se le pasa como segundo parámetro un array de opciones. La opción **allowed_classes** que se usa en este ejemplo, puesta a **false**, hace que no se permita la serialización de objetos de ninguna clase (sólo tipos simples, que componen los atributos de la entidad **Usuario**, en este caso).

CONFIGURAR ORIGEN DE DATOS Y MODO DE AUTENTICACIÓN

- Para indicar a Symfony dónde buscar estos usuarios cuando alguien intente acceder al sistema, añadimos al archivo de configuración config/packages/security.yaml un nuevo proveedor de datos, enlazado a la entidad Usuario:

```
security:
    providers:
        users_in_memory: { memory: null }
        user_provider:
            entity:
                class: App\Entity\Usuario
                property: login
```

Indicamos el nombre de la entidad, y el **nombre del atributo que hace de login.**

- Algo más abajo, establecemos que nos loguearemos mediante un formulario, que se activará con la ruta llamada *login*, la cual definiremos después:

```
security:
    firewalls:
        dev:
        ...
        main:
            anonymous: ~
            form_login:
                provider: user_provider
                login_path: login
                check_path: login
```

En este caso, hemos puesto a null el acceso anónimo (es lo que significa el símbolo "~" junto a anonymous).

Por otra parte, indicamos que tanto para mostrar el formulario como para verificar el **logueo**, se acudirá a la misma ruta **login**, usando como proveedor de usuarios el elemento **user_provider** definido antes.

CONFIGURAR ORIGEN DE DATOS Y MODO DE AUTENTICACIÓN

- Podemos, además, definir qué recursos se van a proteger:

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

En este caso, simplemente se dice que para ir a cualquier ruta (salvo la de login) hay que ser usuario.

Conviene, al menos, especificar un patrón de ruta que excluya al propio formulario de login, ya que de lo contrario entraríamos en un bucle infinito en el que, para acceder al formulario de login, debemos loguearnos con el formulario de login. Por ello, es importante que exista esta primera línea.

- Finalmente, también podemos especificar el sistema de codificado del password. Por ahora no lo vamos a codificar, por eso añadimos este subapartado al final, dentro de la sección security:

```
security:
    …
    encoders:
        App\Entity\Usuario: plaintext
```

DEFINIR LA RUTA Y EL FORMULARIO DE LOGIN

Ahora crearemos un nuevo controlador llamado **LoginController** en nuestra carpeta de **src/Controller**, que defina la ruta /login para mostrar el formulario de login y mostrar error de validación, si es el caso. Puede quedar más o menos así:

```php
<?php
    namespace App\Controller;

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

    class LoginController extends AbstractController
    {
        /**
        * @Route("/login", name="login")
        */
        public function login(AuthenticationUtils $authenticationUtils)
        {
            $error = $authenticationUtils->getLastAuthenticationError();
            $lastUsername = $authenticationUtils->getLastUsername();

            return $this->render('login.html.twig',
                        array('error' => $error, 'lastUsername' => $lastUsername));
        }
    }
?>
```

Estas dos variables entran en juego sólo si se hace un **intento de login fallido.** Recogen el error y el nombre del usuario que se puso, y se envían a la vista por si queremos mostrarlas.

DEFINIR LA RUTA Y EL FORMULARIO DE LOGIN

El formulario de login al que se llama, **login.html.twig**, puede ser algo así:

```
{% extends 'base.html.twig' %}
{% block title %}Contactos{% endblock %}
{% block body %}
    <h1>Login</h1>
    {% if error %}
        <div>{{ error.messageKey }}</div>
    {% endif %}
    <form action="{{ path('login') }}" method="post">
    <label for="username">Login:</label>
    <input type="text" id="username" name="_username" value="{{ lastUsername }}" />
    <label for="password">Password:</label>
    <input type="password" id="password" name="_password" />
    <button type="submit">Entrar</button>
    </form>
{% endblock %}
```

Con esto, se conserva el nick introducido, si falló el login.

Los name deben llamarse así, tal cual: **_username** y **_password**

- **Solo con estos pasos, ya está listo el sistema de autenticación.** Si hay algún error, se registrará en el controlador login y se verá el formulario con el mensaje de error. Si todo es correcto, se redirige **automáticamente** al usuario a la página que solicitó.

ENCRIPTAR LAS CONTRASEÑAS

Es conveniente que las contraseñas de los usuarios registrados no estén en texto plano sin encriptar. Podemos usar, por ejemplo, un algoritmo de encriptación ***bcrypt*** para cifrarlas. Esto requiere modificar dos aspectos:

- Indicar a Symfony que los passwords están encriptados con *bcrypt* para que aplique este algoritmo al encriptar cualquier password, incluido el que introduzca el usuario al loguearse, y así poder comparar si los dos passwords encriptados coinciden. Para hacer esto, debemos editar el archivo de configuración **config/packages/security.yaml** e indicar que la entidad **Usuario** usará el método de encriptación que hayamos escogido:

```
security:
...
    encoders:
        App\Entity\Usuario:
            algorithm: bcrypt
            cost: 12
```

El parámetro **cost** indica cuántas vueltas da el proceso para codificar el password, en un rango entre 4 y 31. Cuanto más largo sea, más costará encriptarlo. Hay otros métodos de encriptación aceptados, como **md5**, **sha256**… aunque según la documentación oficial de Symfony, **bcrypt** es el más recomendable.

ENCRIPTAR LAS CONTRASEÑAS

Para probar esto, podemos codificar manualmente los passwords que tengamos en la BD. Para ello, podemos usar webs como https://bcrypt-generator.com/.

En la sección de **Encrypt**, indicamos en el cuadro String to encrypt el password sin encriptar, y en el cuadro inferior establecemos el coste o número de vueltas. Después pulsamos el botón de Hash! y obtendremos el password encriptado.

ENCRIPTAR LAS CONTRASEÑAS

- Para encriptar de forma automática los passwords de los usuarios cuando se registren usando un formulario de registro, sería así:

```php
use App\Entity\Usuario;
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoderInterface;

public function register(UserPasswordEncoderInterface $encoder)
{
    $usuario = new Usuario();
    // aquí asignamos el resto de atributos del usuario
    $passwordCodificado = $encoder->encodePassword($usuario, $plainPassword);
    $usuario->setPassword($passwordCodificado);
    // guardaríamos en la base de datos, si procede
}
```

Lo que hacemos es acudir a la configuración del archivo **security.yaml** anterior para ver qué codificador se ha establecido, y mediante el objeto de tipo **UserPasswordEncoderInterface** que recibe como parámetro el método, y de su método **encodePassword**, codificar el **password** con ese mismo algoritmo. Dicho método recibe como primer parámetro el **usuario** sobre el que se está trabajando, y como segundo parámetro el **password** a codificar. Una vez codificado, se le asigna al **usuario**, y ya se podría guardar en la base de datos.

TRABAJAR CON ROLES

- En los ejemplos hechos hasta ahora, nos hemos limitado a definir un campo rol en nuestra entidad **Usuario** y almacenar un usuario con rol **ROLE_USER.**
- Todos los roles que definamos en nuestra aplicación deben comenzar con el prefijo **ROLE_** para que Symfony los trate como tales. Existe la posibilidad de configurar esta opción, aunque no lo vamos a estudiar en este curso.
- Symfony permite definir diferentes roles en una aplicación, y establecer una jerarquía entre ellos, de forma que un rol pueda hacer todo lo que hace otro más otras cosas. Además, podemos proteger el acceso a recursos para determinados roles, de forma que sólo ciertos roles puedan acceder.

ASIGNAR ROLES A RECURSOS PROTEGIDOS

- En el caso de que cada zona protegida de nuestra aplicación pueda tener asignados roles diferentes, basta con indicar el rol (o roles entre corchetes) que tienen permiso para cada zona. Por ejemplo, en este caso protegemos el acceso a cualquier ruta que empiece por **/contacto** para que sólo puedan acceder usuarios con rol **ROLE_USER** o **ROLE_MANAGER**, además de la configuración ya establecida en ejemplos previos:

```
security:
...
    access_control:
        - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/contacto, roles: [ROLE_USER, ROLE_MANAGER] }
        - { path: ^/, roles: ROLE_USER }
```

Esta es la que vimos que siempre tiene que estar

Significa que, para poder ir al resto de rutas, tienes que ser ROLE_USER

Significa que, para poder ir a todas las rutas que empiecen por **/contacto**, tienes que ser **ROLE_USER** o **ROLE_MANAGER**

Se suele hacer así. Primero se concretan algunas rutas en concreto (para administradores, por ejemplo) y se termina usando ^/ para referirse al resto de rutas. Es como ir de más concreto a más general.

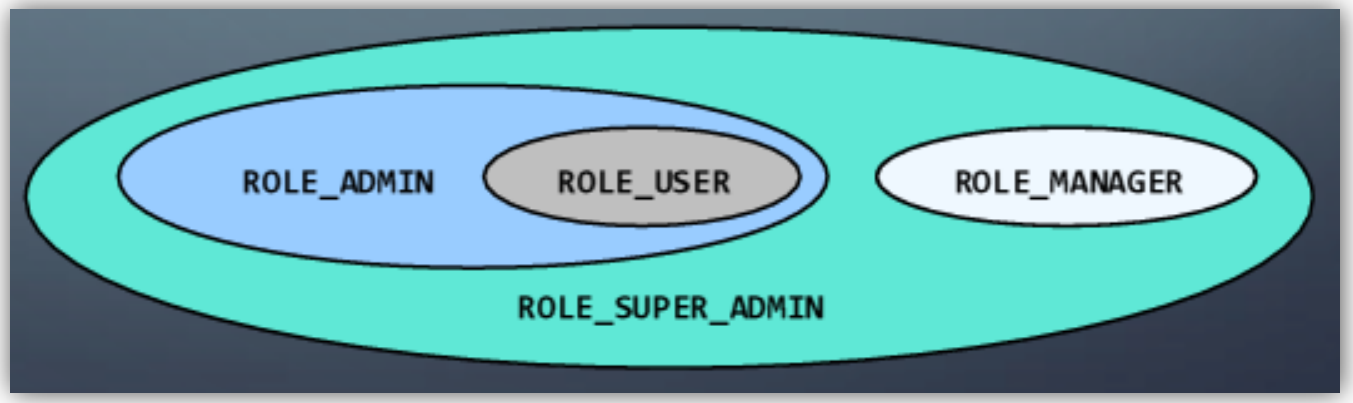
Para poner en una línea varias rutas, se haría así:
- { path: ^/**(una|otra|otramas)**, roles: … }

ESTABLECER JERARQUÍAS ENTRE ROLES

- Puede ser necesario también establecer una jerarquía entre roles, de forma que, si se tiene un rol de nivel superior, se tendrá acceso a todos los recursos que exijan un rol de nivel inferior. Para hacer esto, añadimos un subapartado ***role_hierarchy*** en nuestra sección ***security*** de **config/packages/security.yaml**.

```
security:
...
    role_hierarchy:
        ROLE_ADMIN: ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_MANAGER]
```

En este ejemplo, el rol **ROLE_ADMIN** contiene a su vez a **ROLE_USER,** y el **ROLE_SUPERADMIN** contiene tanto a **ROLE_ADMIN** (y, por tanto, a **ROLE_USER**), como a **ROLE_MANAGER.**



COMPROBAR ROLES DESDE CONTROLADORES Y VISTAS

- Se puede forzar, dentro de un controlador, una comprobación de seguridad para saber si el usuario registrado tiene cierto rol, o simplemente si se ha autenticado.

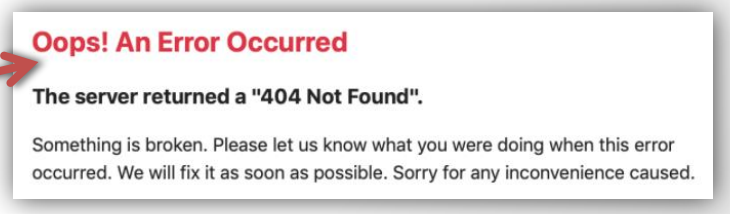
```php
/**
* @Route("/administrar", name="administrar")
*/
public function administrar()
{
    $this->denyAccessUnlessGranted('ROLE_ADMIN', null, 'Acceso restringido a administradores');
    // Resto del código del controlador
}

/**
* @Route("/otro", name="otro")
*/
public function otroControlador()
{
    $this->denyAccessUnlessGranted('IS_AUTHENTICATED_FULLY');
    // Resto del código del controlador
}
```

Esta línea deniega el acceso al controlador a menos que el usuario sea ROLE_ADMIN

Esta línea impide el acceso al controlador a menos que el usuario esté autenticado (sea cual sea su rol)

Si **security.yaml** está bien hecho, no hace falta esto. No obstante, podría añadirse como medida "extra" de seguridad

En cualquier caso, el método **denyAccessUnlessGranted** provoca que:
- Si el usuario aún no se ha autenticado, se le redirige a la página de **login**
- Si se ha autenticado, pero no tiene el rol requerido, se genera una página de error **HTTP 403** (acceso prohibido). Esta página se puede personalizar (más información aquí).



COMPROBAR ROLES DESDE CONTROLADORES Y VISTAS

También es posible comprobar un determinado rol o autenticación desde el código de una plantilla Twig, mediante la función **is_granted.**

```
{% if is_granted('ROLE_ADMIN') %}
    ...
{% endif %}
```

Esto es muy útil, por ejemplo para mostrar en el menú principal de la web sólo los enlaces que correspondan a cada rol. También es muy útil, dentro de cierta sección, para mostrar/ocultar botones o secciones a ciertos roles.

OBTENER EL OBJETO USUARIO

- Es probable que, dentro de un controlador, necesitemos acceder al objeto del usuario logueado. Por ejemplo, en una web de alquiler de coches, para actualizar la BD cuando el usuario reserve un vehículo.

```
/**
* @Route("/reservarCoche/{numcar}", name="reservarCoche")
*/
public function reservarCoche($numcar)
{
    $nombreUsuario = $this->getUser()->getUserName();
    // Aquí llamaríamos a un método del repositorio pasándole $numcar y $nombreUsuario
    para hacer la reserva en la BD.
}
```

- Si queremos acceder al objeto usuario desde una plantilla Twig, por ejemplo para darle la bienvenida, se hace con **app.user** de la siguiente manera:

```
Bienvenido/a, {{ app.user.username }}
```

CIERRE DE SESIÓN (LOGOUT)

Fácilmente podemos configurarlo a través de archivos YAML, mediante dos pasos:

1. Establecemos que cuando se salga de la aplicación se redirigirá, por ejemplo, a la raíz de la misma. Esto se hace en el archivo **config/packages/security.yaml**:

```
firewalls:
    ...
    main:
        ...
        logout:
            path: /logout
            target: /
```

2. Definimos una ruta en el archivo **config/routes/routes.yaml** que asocie el nombre "logout" (o como lo queramos llamar) con la ruta definida anteriormente (en este caso, la ruta `/logout`):

```
logout:
    path: /logout
```

Este archivo **routes.yaml** no lo habíamos usado hasta ahora, gracias a que definíamos las rutas como anotaciones encima de los controladores. Sin embargo, para algo tan simple como es el logout, optaremos por esta opción, para no tener que crearle siquiera un controlador.

Hecho esto, sólo nos falta un enlace en las plantillas para poder cerrar sesión:

```
<a href="{{ path('logout') }}">Cerrar sesión</a>
```

EJERCICIO 1

- Crea en tu proyecto *libros* la misma entidad **Usuario** del ejemplo de *contactos*, implementando las interfaces necesarias para que funcione dentro del sistema de seguridad de Symfony y realizando la migración correspondiente a la BD.

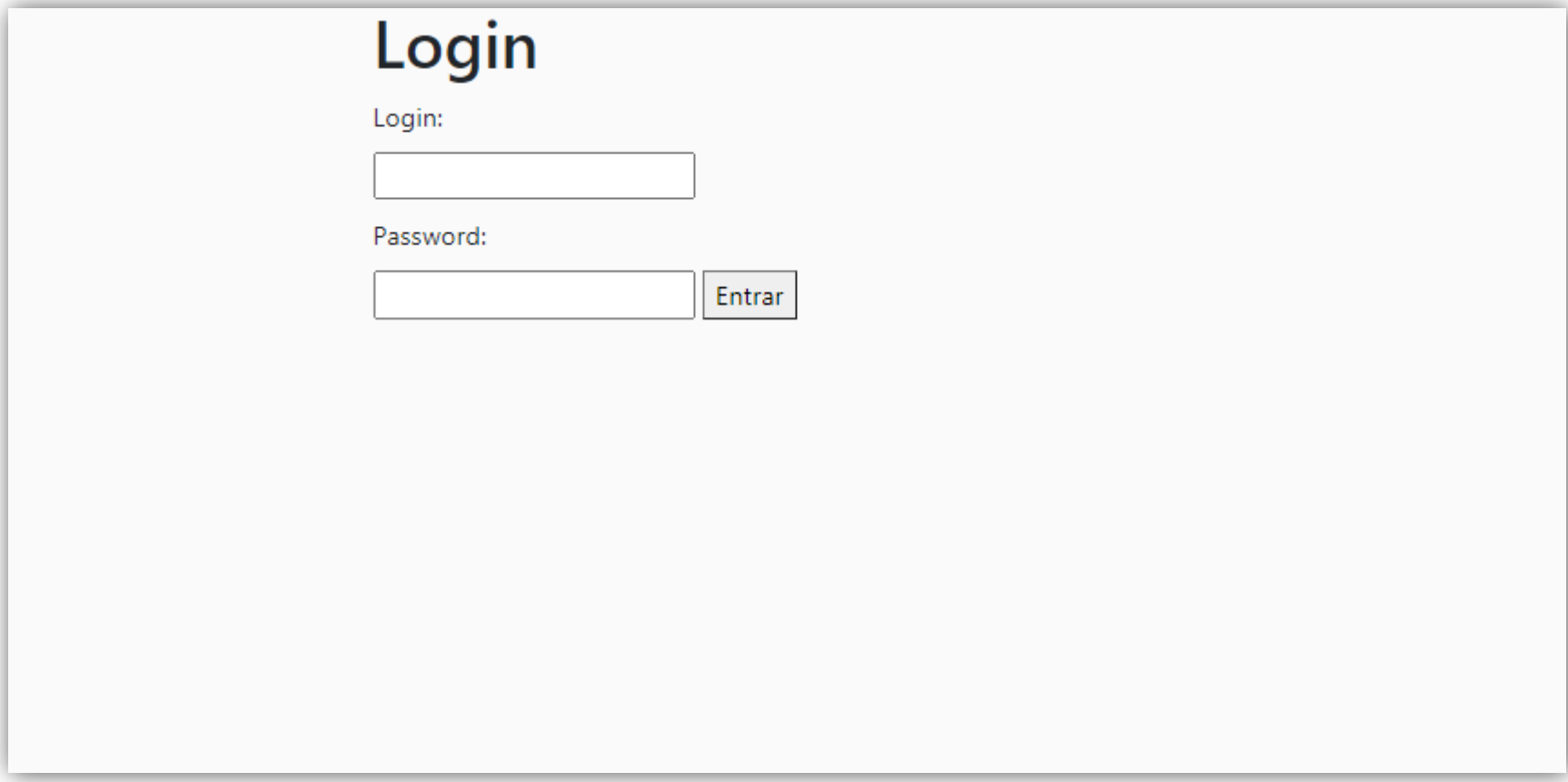
	#	Nombre	Tipo	Cotejamiento	Atributos	Nulo
☐	1	id 🔑	int(11)			No
☐	2	login	varchar(100)	utf8mb4_unicode_ci		No
☐	3	password	varchar(100)	utf8mb4_unicode_ci		No
☐	4	email	varchar(100)	utf8mb4_unicode_ci		No
☐	5	rol	varchar(20)	utf8mb4_unicode_ci		No

- Define un formulario de login (**login.html.twig**) como el del ejemplo de contactos, y configura el archivo **config/packages/security.yaml** para que use dicho formulario, bajo la ruta **/login** (define también el controlador asociado a esa ruta en **LoginController.php**).
- Añade manualmente desde *phpmyadmin* dos usuarios con los roles *user* y *admin*:

id	login	password	email	rol
1	pepe	$2y$12$yTS424V9hXcclE53uHWkaebJqGfWDKogPhF.Z0c.xTn...	pepe@gmail.com	ROLE_USER
2	juan	$2y$12$mOh0ue.2fnIn5Ae06cHclOUcuhvrrvWfQ6meVSk.B3S...	juan@gmail.com	ROLE_ADMIN

- Como ves, vamos a tener sólo dos roles:
  - ✓ **ROLE_USER**
  - ✓ **ROLE_ADMIN** (hará todo lo que puede hacer **ROLE_USER.** Debe especificarse)
- Deberás proteger algunos recursos de la aplicación:
  - ✓ Los invitados sólo podrán ver la página para loguearse.
  - ✓ Los **ROLE_USER** podrán hacer lo que tenemos implementado, menos estas cosas:
    - Insertar libros
    - Editar libros
    - Eliminar libros
  - ✓ Los **ROLE_ADMIN** podrán hacer todo lo que tenemos implementado.
- Por último, ten en cuenta estas dos cosas:
    - Los apartados no permitidos no sólo deben ocultarse visualmente, sino que hay que prohibir el acceso.
    - Se recomienda mirar con atención las capturas de las diapositivas siguientes.

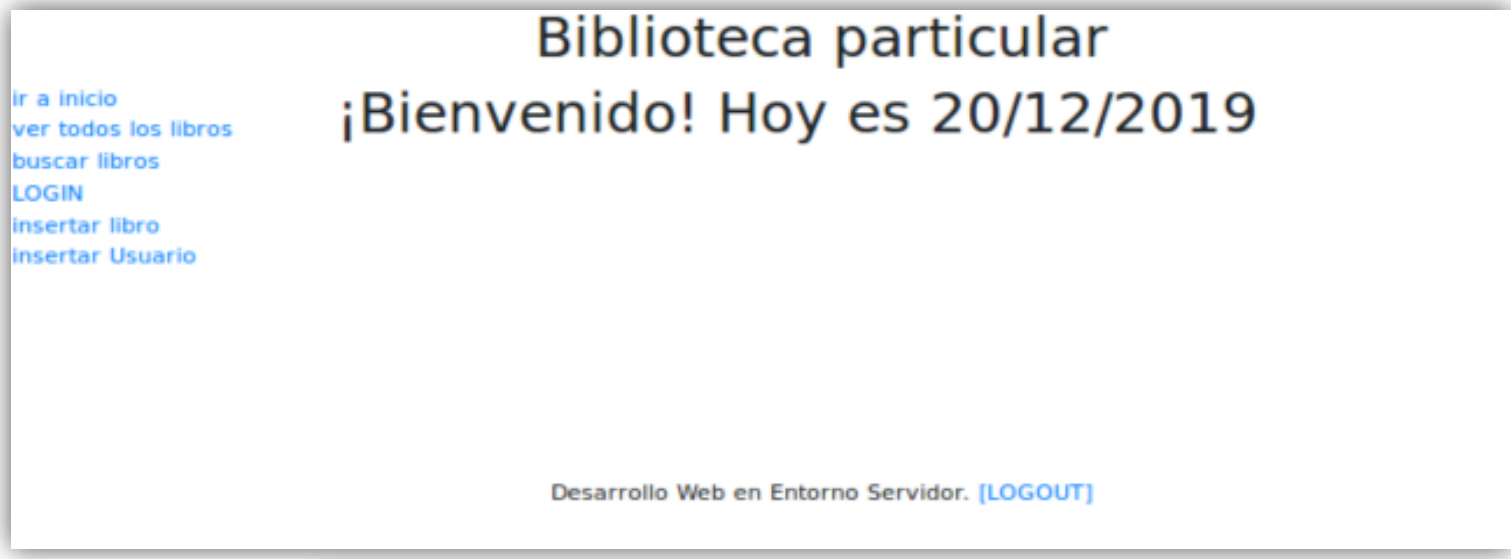
- Entrando en **/** nos redirige a **/login**



- **Logueado como ROLE_USER:**

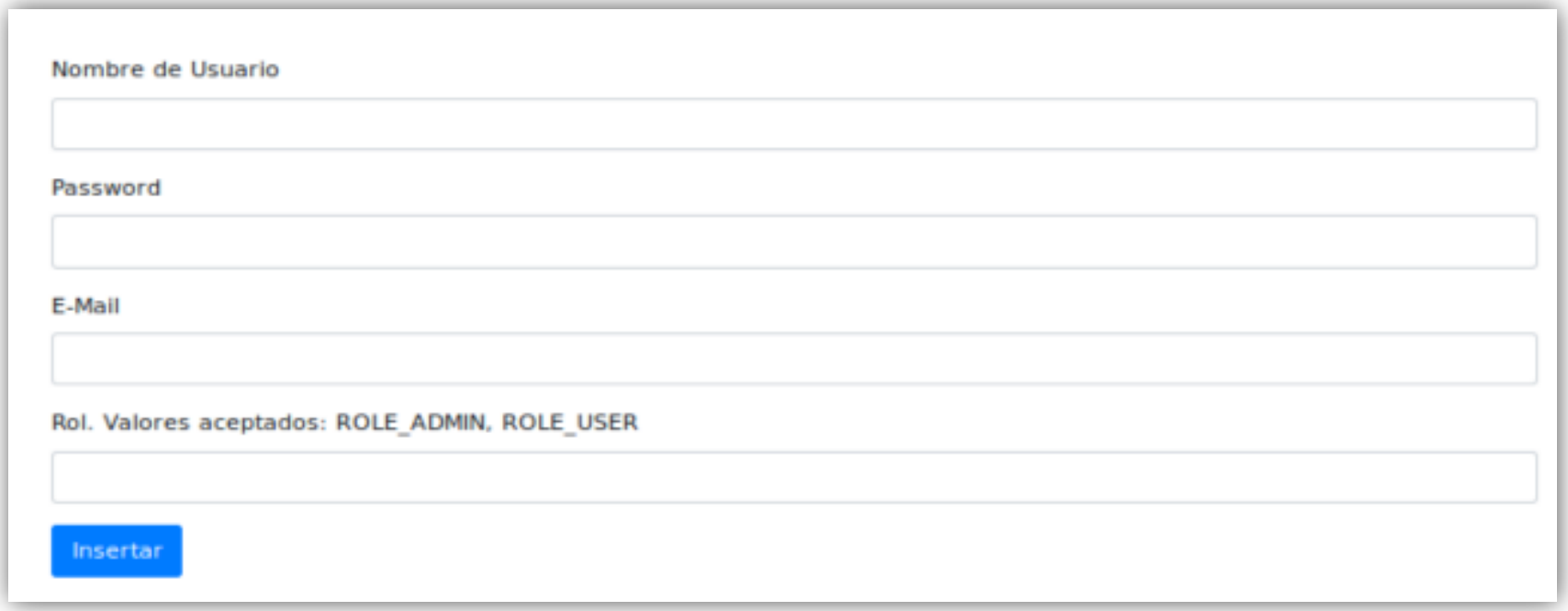
- ✓ El *user* solo puede visualizar los libros y utilizar el buscador.

- **Logueado como ROLE_ADMIN:**



- ✓ El *admin* puede añadir editar y eliminar libros.
- ✓ El *admin* puede insertar nuevos usuarios.

- Crea el fichero **UsuarioController.php** con controlador de ruta **/usuario/nuevo**, y nombre **nuevo_usuario**.
- El usuario **ROLE_ADMIN** podrá dar de alta a otros usuarios.



libros

Archivo Editar Ver Marcadores Ir Herramientas Ayuda

/home/alumno/symfony/libros

Lugares

Carpeta personal
Escritorio
Papelera
Aplicaciones
Documentos
Música
Imágenes
Videos
Descargas

bin
config
public
src
templates
tests
translations
var
vendor
composer.json
composer.lock
phpunit.xml.dist
symfony.lock

13 elementos (3 ocultos)

libros.zip

Se entrega la carpeta del proyecto **libros** comprimida, pero **sin la carpeta vendor**

Podéis sacarla y meterla arrastrando. O eliminarla y luego recuperarla desde el terminal, ubicándote en la carpeta del proyecto y ejecutando `composer install`