Seguridad - Autenticación

El sistema de seguridad de symfony es muy potente, pero también puede llegar a ser muy confuso.

Para instalar el componente de seguridad con Symfony Flex, en caso de no tenerlo ya instalado, hay que ejecutar

> *composer require security*

La seguridad se configura en el archivo security.yml. Por defecto, tiene el siguiente aspecto:

```
# app/config/security.yml
security:
    providers:
        in_memory:
            memory: ~

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        main:
            anonymous: ~
```

Referencia del fichero de configuración security.yaml:

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

Autenticación

Vamos a empezar con un ejemplo básico de seguridad: Vamos a limitar el acceso a algunas páginas de nuestra aplicación y a pedir autenticación básica HTTP.

```
security:
    providers:
        in_memory:
            memory: ~

    firewalls:
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false

        main:
            anonymous: ~
            http_basic: ~

    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN }
        - { path: ^/alumno, roles: ROLE_ADMIN }
        - { path: ^/asignatura, roles: ROLE_USER }
```

En la sección access_control, restringimos el acceso a determinadas

rutas de forma que únicamente puedan acceder aquellos usuarios con un rol determinado.

En el ejemplo anterior, solamente los usuarios con rol ROLE_ADMIN pueden acceder a las rutas que empiezan por /admin, /alumno y /asignatura.

Si un usuario intenta entrar en dichas páginas y no tiene acceso, se le mostrará el formulario de login HTTP básico.

En la sección providers, configuramos el sistema o sistemas proveedores de usuarios. Vamos a ver un ejemplo sencillo del proveedor *in_memory*.

```
    providers:
        in_memory:
            memory:
                users:
                    carlos:
                        password: pass
                        roles: 'ROLE_USER'
                    admin:
                        password: word
                        roles: 'ROLE_ADMIN'
```

Este proveedor tiene dos usuarios:

- usuario: carlos, contraseña: pass, con rol ROLE_USER
- usuario: admin, contraseña: word, con rol ROLE_ADMIN

Ya solamente falta informar al sistema de seguridad de cuál es el algoritmo utilizado para codificar las contraseñas. Como en este caso las contraseñas están sin codificar, indicaremos que el algoritmo es *plaintext*.

```
    encoders:
        Symfony\Component\Security\Core\User\User: plainte
xt
```

Ya podemos probar el login.

Si probamos a entrar en */asignaturas* con el usuario *carlos* veremos que nos aparece un error HTTP 403 Forbidden. Es lo esperado, ya que el usuario *carlos* no tiene permisos para acceder a esa página.

Podemos observar también que en la barra de depuración aparece el nombre de usuario y en el profiler, información detallada de los aspectos de seguridad de esta petición.

Si probamos a entrar en */asignaturas* con el usuario *admin* veremos que sí que nos deja acceder.

Cambiar el algoritmo de codificación de contraseñas

Vamos a cambiar el algoritmo de codificación de contraseñas de texto plano a *bcrypt* .

```
    encoders:
        Symfony\Component\Security\Core\User\User:
            algorithm: bcrypt
            cost: 12
```

Existe un comando en symfony que codifica una contraseña utilizando el algoritmo configurado:

> *bin/console security:encode-password*

Más ejemplos de encoders:

```
    encoders:
        # Examples:
        App\Entity\User1: sha512
        App\Entity\User2:
            algorithm:           sha512
            encode_as_base64:    true
            iterations:          5000

        # PBKDF2 encoder
        # see the note about PBKDF2 below for details on s
```

```
ecurity and speed
        App\Entity\User3:
            algorithm:            pbkdf2
            hash_algorithm:       sha512
            encode_as_base64:     true
            iterations:           1000
            key_length:           40

        # Example options/values for what a custom encoder
 might look like
        App\Entity\User4:
            id:                   App\Security\MyPasswordE
ncoder

        # BCrypt encoder
        # see the note about bcrypt below for details on s
pecific dependencies
        App\Entity\User5:
            algorithm:            bcrypt
            cost:                 13

        # Plaintext encoder
        # it does not do any encoding
        App\Entity\User6:
            algorithm:            plaintext
            ignore_case:          false

        # Argon2i encoder
```

```
        Acme\DemoBundle\Entity\User6:
            algorithm:            argon2i
```

Entity Provider

Vamos ahora a cambiar el provider in_memory por uno de base de datos:

Lo primero que necesitamos es nuestra entidad de usuarios

```
namespace AppBundle\Entity;

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

/**
 * @ORM\Table(name="app_users")
 * @ORM\Entity(repositoryClass="AppBundle\Repository\UserR
epository")
 */
class User implements UserInterface, \Serializable
{
    /**
     * @ORM\Column(type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
```

```
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=25, unique=true)
     */
    private $username;

    /**
     * @ORM\Column(type="string", length=64)
     */
    private $password;

    /**
     * @ORM\Column(type="string", length=60, unique=true)
     */
    private $email;

    /**
     * @ORM\Column(name="is_active", type="boolean")
     */
    private $isActive;

    public function __construct()
    {
        $this->isActive = true;
        // si necesitáramos un "salt" podríamos hacer algo
así
```

```
        // $this->salt = md5(uniqid('', true));
    }

    public function getUsername()
    {
        return $this->username;
    }

    public function getSalt()
    {
        // El método es necesario aunque no utilicemos un
"salt"
        return null;
    }

    public function getPassword()
    {
        return $this->password;
    }

    public function getRoles()
    {
        return array('ROLE_USER');
    }

    public function eraseCredentials()
    {
    }
```

```
    public function serialize()
    {
        return serialize(array(
            $this->id,
            $this->username,
            $this->password,
            // see section on salt below
            // $this->salt,
        ));
    }

    public function unserialize($serialized)
    {
        list (
            $this->id,
            $this->username,
            $this->password,
            // see section on salt below
            // $this->salt
        ) = unserialize($serialized);
    }
}
```

Y ahora actualizamos la base de datos:

> *php bin/console doctrine:schema:update --force*

Una clase *User* debe implementar las interfaces UserInterface y Serializable.

Como consecuencia de implementar la interfaz UserInterface tenemos que crear los siguientes métodos:

- getRoles()
- getPassword()
- getSalt()
- getUsername()
- eraseCredentials()

Y como consecuencia de implementar Serializable, tenemos que crear los siguientes métodos:

- serialize()
- unserialize()

Al final de cada petición el objeto User es serializado y metido en la sesión. En la siguiente petición, se deserializa. Symfony hace dichas operaciones llamando a los métodos serialize() y unserialize().

Ya solamente queda configurar el security.yml para que utilice un provider basado en nuestra entidad

```
# app/config/security.yml
security:
    encoders:
        AppBundle\Entity\User:
            algorithm: bcrypt

    # ...

    providers:
        mi_poveedor:
            entity:
                class: AppBundle:User
                property: username

    firewalls:
        main:
            pattern:    ^/
            http_basic: ~
            provider: mi_poveedor

    # ...
```

AdvancedUserInterface

En vez de extender de UserInterface, podemos extender de AdvancedUserInterface.

Para ello tenemos que definir los siguientes métodos:

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

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

class User implements AdvancedUserInterface, \Serializable
{
    // ...

    public function isAccountNonExpired()
    {
        return true;
    }

    public function isAccountNonLocked()
    {
        return true;
    }

    public function isCredentialsNonExpired()
    {
        return true;
    }

    public function isEnabled()
    {
```

```
        return $this->isActive;
    }

    // serialize and unserialize must be updated - see bel
ow
    public function serialize()
    {
        return serialize(array(
            // ...
            $this->isActive
        ));
    }
    public function unserialize($serialized)
    {
        list (
            // ...
            $this->isActive
        ) = unserialize($serialized);
    }
}
```

- isAccountNonExpired(): comprueba si la cuenta de usuario ha caducado
- isAccountNonLocked(): comprueba si el usuario está bloqueado
- isCredentialsNonExpired() comprueba si la contrasea ha caducado;
- isEnabled() comprueba si el usuario est habilitado.

Si cualquiera de estos métodos devuelve false, el usuario no podrá hacer login.

Según cuál de estos métodos devuelva falso, Symfony generará un mensaje diferente.

Configurar múltiples providers

Es posible configurar múltiples providers. En caso de que un firewall no especifique qué provider va a utilizar, utilizará el primero de ellos.

```
# app/config/security.yml
security:
    providers:
        chain_provider:
            chain:
                providers: [in_memory, user_db]
        in_memory:
            memory:
                users:
                    foo: { password: test }
        user_db:
            entity: { class: AppBundle\Entity\User, proper
ty: username }

    firewalls:
        secured_area:
```

```
            # ...
            pattern: ^/
            provider: user_db
            form_login: ~
```

Symfony dispone de 4 providers ya programados:

- memory
- entity
- ldap
- chain

El provider *chain* no es un provider en sí, sino que sirve para especificar una cadena de providers.

Autenticación con formulario de login

Vamos a cambiar ahora el método de login http_basic por un formulario de login.

```
# app/config/security.yml
security:
    # ...

    firewalls:
        main:
```

```
            anonymous: ~
            form_login:
                login_path: login
                check_path: login
                default_target_path: after_login_route_nam
e
                always_use_default_target_path: true
```

Con esto decimos a symfony que vamos a utilizar un formulario de login, y que redirija a la ruta de nombre “login” cuando sea necesario identificar a un usuario.

Los siguiente es crear una acción y una plantilla para esa ruta:

```
use Symfony\Component\Security\Http\Authentication\Authent
icationUtils;

public function loginAction(Request $request, Authenticati
onUtils $authUtils)
{
    // get the login error if there is one
    $error = $authUtils->getLastAuthenticationError();

    // last username entered by the user
    $lastUsername = $authUtils->getLastUsername();

    return $this->render('security/login.html.twig', array
```

```
(
        'last_username' => $lastUsername,
        'error'         => $error,
    ));
}
```

```
{% if error %}
    <div>{{ error.messageData }}</div>
{% endif %}

<form action="{{ path('login') }}" method="post">
    <label for="username">Usuario:</label>
    <input type="text" id="username" name="_username" valu
e="{{ last_username }}" />

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

    {#
        Si queremos controlar la url a la que se redirigir
á el usuario después de hacer login
        <input type="hidden" name="_target_path" value="/a
ccount" />
    #}

    <button type="submit">Login</button>
```

```
</form>
```

Roles

Al hacer login, el usuario recibe un conjunto específico de roles (por ejemplo: ROLE_ADMIN).

Los roles deben empezar con el prefijo *ROLE_*.

Un usuario autenticado debe de tener al menos un rol.

Es posible establecer una jerarquía de roles:

```
security:
    # ...

    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER
        ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWI
TCH]
```

Los pseudo-roles

Symfony tiene 3 pseudo-roles (atributos), que no son roles, pero actúan como si lo fueran:

- IS_AUTHENTICATED_ANONYMOUSLY: Todos los usuarios tienen este atributo, estén logeados o no
- IS_AUTHENTICATED_REMEMBERED: Todos los usuarios
- logeados tienen este atributo
  IS_AUTHENTICATED_FULLY: Todos los usuarios logeados excepto los que están logeados a través de una “remember me cookie”.

Seguridad - Autorización

El proceso de autorización consiste en añadir código para que un recurso requiera un *atributo* específico (un rol u otro tipo de atributo) para acceder a dicho recurso.

Añadir código para denegar el acceso a un recurso puede hacerse bien mediante la sección *access_control* del security.yml o bien a través del servicio *security.authorization_checker*.

Access control

En el access control, además de la url, se puede configurar accesos por IP, host name o métodos HTTP.

También se puede utilizar la sección *access_control* para redireccionar al usuario al protocolo *https*

Ejemplos:

```
security:
    # ...
    access_control:
        - { path: ^/admin, roles: ROLE_ADMIN, ip: 127.0.0.
1 }
        - { path: ^/admin, roles: ROLE_ADMIN, host: symfon
y\.com$ }
```

```
        - { path: ^/admin, roles: ROLE_ADMIN, methods: [PO
ST, PUT] }
        - { path: ^/admin, roles: ROLE_ADMIN }
```

Primero Symfony busca el match correspondiente según las coincidencias de:

- path
- ip
- host
- methods

Una vez vista cuál es la coincidencia, permite o deniega el acceso, según se cumplan las condiciones de:

- roles: si el usuario no tiene este rol, se le deniega el acceso
- allow_if: si la expresión evaluada devuelve *false* se le deniega el acceso

```
security:
    # ...
    access_control:
        -
            path: ^/_internal/secure
            allow_if: "'127.0.0.1' == request.getClientIp(
) or has_role('ROLE_ADMIN')"
```

- requires_channel: si el protocolo (canal) de la petición no coincide con el indicado, se le redirige al indicado.

```
security:
    # ...
    access_control:
        - { path: ^/cart/checkout, roles: IS_AUTHENTICATED
_ANONYMOUSLY, requires_channel: https }
```

Si el acceso resulta denegado y el usuario no está autenticado, se le redirige al sistema de autenticación configurado.

Si el acceso resulta denegado y ya estaba autenticado, se le muestra una página de 403 acceso denegado.

El servicio Authorization_checker

La forma de añadir código de denegación de acceso a través del servicio *security.authorization_checker*
son las siguientes:

A) En los controladores:

```
public function helloAction($name)
{
    // The second parameter is used to specify on what obj
```

```
ect the role is tested.
    $this->denyAccessUnlessGranted('ROLE_ADMIN', null, 'Un
able to access this page!');

    // Old way :
    // if (false === $this->get('security.authorization_ch
ecker')->isGranted('ROLE_ADMIN')) {
    //     throw $this->createAccessDeniedException('Unabl
e to access this page!');
    // }

    // ...
}
```

Gracias al bundle SensioFrameworkExtraBundle, se puede hacer lo mismo con anotaciones:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Secur
ity;

/**
 * @Security("is_granted('ROLE_ADMIN')")
 */
public function helloAction($name)
{
    // ...
}
```

Incluso se puede poner a nivel de la clase controladora

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Secur
ity;

/**
 * @Security("is_granted('ROLE_ADMIN')")
 * @Route("/asignaturas")
 */
class AsignaturasController extends Controller
{

}
```

B) En las plantillas

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

C) En los servicios

```
// src/AppBundle/Newsletter/NewsletterManager.php

// ...
use Symfony\Component\Security\Core\Authorization\Authoriz
```

```
ationCheckerInterface;
use Symfony\Component\Security\Core\Exception\AccessDenied
Exception;

class NewsletterManager
{
    protected $authorizationChecker;

    public function __construct(AuthorizationCheckerInterf
ace $authorizationChecker)
    {
        $this->authorizationChecker = $authorizationChecke
r;
    }

    public function sendNewsletter()
    {
        if (!$this->authorizationChecker->isGranted('ROLE_
NEWSLETTER_ADMIN')) {
            throw new AccessDeniedException();
        }

        // ...
    }

    // ...
}
```

El servicio Security

En las nuevas versiones de Symfony, el servicio de seguridad se llama ahora Security.

```
// src/AppBundle/Newsletter/NewsletterManager.php

// ...
use Symfony\Component\Security\Core\Exception\AccessDenied
Exception;
use Symfony\Component\Security\Core\Security;

class NewsletterManager
{
    protected $security;

    public function __construct(Security $security)
    {
        $this->security = $security;
    }

    public function sendNewsletter()
    {
        if (!$this->security->isGranted('ROLE_NEWSLETTER_A
DMIN')) {
            throw new AccessDeniedException();
        }
```

```
            // ...
        }

        // ...
}
```

Acceso al objeto User

Tras la autenticación, el objeto User asociado al usuario actual se puede obtener a través del servicio *security.token_storage*.

En un controlador, podemos tener acceso fácilmente al objeto User gracias a la inyección de dependencias.

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

public function index(UserInterface $user)
{
    //...
}
```

De qué tipo de clase sea nuestro objeto *$user* dependerá de nuestro *user provider*.

Si nuestra clase controladora extiende de Controller se puede acceder también al usuario con $this->getUser().

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

public function index()
{
    $user = $this->getUser();
```

```
    //...
}
```

En twig, podemos acceder al objeto user con app.user

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

Las anotaciones @IsGranted y @Security

Las anotaciones @IsGranted y @Security restringen el acceso a los controladores:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Secur
ity;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\IsGra
nted;

class PostController extends Controller
{
    /**
     * @IsGranted("ROLE_ADMIN")
     *
     * or use @Security for more flexibility:
     *
     * @Security("is_granted('ROLE_ADMIN') and is_granted(
'ROLE_FRIENDLY_USER')")
     */
    public function indexAction()
    {
        // ...
    }
}
```

@IsGranted

La anotación @IsGranted() es muy simple de utilizar. Se utiliza para restringir roles o para restricciones basadas en Voters:

```
/**
 * @Route("/posts/{id}")
 *
 * @IsGranted("ROLE_ALUMNO")
 * @IsGranted("nota_ver", subject="nota")
 */
public function verAction(Nota $nota)
{
}
```

Para acceder a una acción hay que pasar todas las restricciones.

La anotación @IsGranted permite también personalizar el statusCode y el mensaje de error.

```
/**
 * Will throw a normal AccessDeniedException:
 *
 * @IsGranted("ROLE_ADMIN", message="No access! Get out!")
 *
 * Will throw an HttpException with a 404 status code:
 *
```

```
     * @IsGranted("ROLE_ADMIN", statusCode=404, message="Post
not found")
     */
    public function indexAction(Post $post)
    {
    }
```

@Security

La anotación @Security es más flexible que @IsGranted: permite crear expresiones con lógica personalizada:

```
    /**
     * @Security("is_granted('ROLE_ALUMNO') and is_granted('ve
r', nota)")
     */
    public function verAction(Nota $nota)
    {
        // ...
    }
```

Las expresiones pueden utilizar todas las funciones admitidas en la sección access_control del security.yaml, además de la función is_granted().

Las expresiones tienen acceso a las siguientes variables:

- token: El token de seguridad actual;
- user: El objeto usuario actual;
- request: La instancia de la request;
- roles: Los roles del usuario;
- y todos los atributos de la request.

Se puede lanzar una excepción Symfony\Component\HttpKernel\Exception\HttpException exception en vez de una excepción Symfony\Component\Security\Core\Exception\AccessDeniedException using utilizando el statusCode 404:

```
/**
 * @Security("is_granted('ver', nota)", statusCode=404)
 */
public function verAction(Nota $nota)
{
}
```

También se puede personalizar el mensaje de error:

```
/**
 * @Security("is_granted('ver', nota)", statusCode=404, me
ssage="Resource not found.")
 */
public function verAction(Nota $nota)
{
```

```
}
```

Las anotaciones @IsGranted y @Security se pueden utilizar a nivel de clase para proteger todas las acciones de la clase controladora.

Voters

Los voters son clases que deciden si un usuario puede acceder a un recurso o no.

Cada vez que se llama al método **isGranted()** o al método **denyAccessUnlessGranted()** Symfony hace una llamada a cada clase voter que haya registrada en el sistema.

Cada uno de los voters decidirá si permite al usuario realizar la acción, si le deniega realizarla o si se abstiene de decidir nada. Symfony recoge la respuesta de todos los Voters y toma la decisión final en base a la estrategia configurada.

La clase Voter

Un voter personalizado necesita implementar VoterInterface o extender Voter.

```
abstract class Voter implements VoterInterface
{
    abstract protected function supports($attribute, $subj
ect);
    abstract protected function voteOnAttribute($attribute
, $subject, TokenInterface $token);
}
```

Nuestro voter será por lo tanto una clase con dos métodos:

- supports($attribute, $subject)
- voteOnAttribute($attribute, $subject, TokenInterface $token)

Vamos a suponer que tenemos una aplicación que gestiona eventos, y que los usuarios solamente pueden editar los eventos de los que son creadores.

Es decir, cualquier usuario puede editar eventos, pero solamente los suyos.

```
    /**
     * @Route("/eventos/{id}/edit", name="eventos_edit")
     */
    public function editAction($id)
    {

        $evento = ...;

        // chequear acceso de edición del evento
        $this->denyAccessUnlessGranted('edit', $evento);

        // ...
    }
```

El método denyAccessUnlessGranted() (y pasaría también con isGranted()) llama al sistema de voters. Ahora mismo no hay vot

puedan juzgar si el usuario puede editar

el evento, así que vamos a crear uno.

```
// src/AppBundle/Security/EditarEventoVoter.php
namespace AppBundle\Security;

use AppBundle\Entity\Evento;
use AppBundle\Entity\User;
use Symfony\Component\Security\Core\Authentication\Token\T
okenInterface;
use Symfony\Component\Security\Core\Authorization\Voter\Vo
ter;

class EditarEventoVoter extends Voter
{
    protected function supports($atributo, $entidad)
    {
        // Este voter sólo toma decisiones sobre editar ob
jetos Evento
        if ($entidad instanceof Evento && $atributo == 'ed
it') {
            return true;
        }

        return false;
    }

    protected function voteOnAttribute($atributo, $entidad
```

```
, TokenInterface $token)
    {
        $user = $token->getUser();

        //Gracias al método supports ya sabemos que $entid
ad es un Evento
        if($user->getId() == $entidad->getCreador()->getId
()){
          return true;
        }

        return false;
    }
}
```

El método supports()

La llamada a *$this->denyAccessUnlessGranted('edit', $evento)* admite dos parámetros.
Estos dos parámetros se le pasan a la función supports del voter.

Este método debe devolver true o false. Si devuelve false, el voter se *abstiene* de tomar ninguna decisión y el sistema de seguridad de symfony lo ignora.

Si por el contrario, el método supports devuelve true, entonces Symfony llamará al método voteOnAttribute().

El método voteOnAttribute()

El objetivo de este método también es muy simple. Si devuelve true, Symfony permitirá al usuario realizar la acción. Si devuelve false, Symfony denegará al usuario realizar la acción.

A este método le llegan dos parámetros con los dos valores con los que se llamó a $this->denyAccessUnlessGranted('edit', $evento) y un tercer parámetro con acceso al objeto user.

Si es necesario, se puede utilizar la inyección de dependencias para acceder a cualquier otro servicio con lo que en un voter tenemos acceso a cualquier elemento de nuestra aplicación que necesitemos para tomar la decisión.

Configurar el voter

Para inyectar el voter en el security layer debes declararlo como service y ponerle el tag security.voter:

```
# app/config/services.yml
services:
    app.post_voter:
        class: AppBundle\Security\EditarEventoVoter
        tags:
            - { name: security.voter }
        # pequeña mejora de rendimiento
```

```
        public: false
```

Este paso no será necesario si tenemos el autoconfigure a true.

Estrategia de decisión

Normalmente tendremos un único voter.

Otras veces podemos tener varios voters, pero solamente un voter votará en cada ocasión y el resto se abstendrán.

Y en raras ocasiones tendremos varios voters tomando una misma decisión.

Por defecto, cuando varios voters tienen que votar si permiten o no un acceso, basta con que uno de ellos lo permita, para que el usuario logre el acceso.

Sin embargo se puede cambiar este comportamiento, llamado “strategy” desde el security.yml. Hay 3 posibles estrategias para elegir:

- affirmative (por defecto). Otorga acceso tan pronto como haya un voter que permita acceso.
- consensus. Otorga acceso si hay más voters garantizando acceso que denegándolo.
- unanimous. Sólo otorga acceso una vez que todos los voters garantizan acceso.

```
# app/config/security.yml
security:
    access_decision_manager:
        strategy: unanimous
```

Soporte para ACL

El soporte para ACL ha sido eliminado en Symfony 4.0. Si se quiere trabajar con ACL existe un bundle preparado para ello: https://github.com/symfony/acl-bundle

allow_if_all_abstain

En caso de que todos los voters se abstengan, el comportamiento por defecto es denegar el acceso al usuario. Se puede configurar este comportamiento con la opción **allow_if_all_abstain**.

```
security:
    access_decision_manager:
        strategy: unanimous
        allow_if_all_abstain: true
```

allow_if_equal_granted_denied

En la estrategia *consensus*, en caso de empate entre voters que permitan el acceso y voters que denieguen el acceso, por defecto se permite el acceso. Se puede cambiar este comportamiento con la opción **allow_if_equal_granted_denied**

```
security:
    access_decision_manager:
        strategy: consensus
        allow_if_equal_granted_denied: false
```

Logout

EL logout no es necesario programarlo, solamente configurarlo.

Primero definimos una ruta de logout en el firewall:

```
# app/config/security.yml
security:
    # ...

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

Y añadimos la ruta al routing

```
# app/config/routing.yml
logout:
    path: /logout
```

Pero NO hay que crear ningún controlador que haga nada.

Cuando un usuario acceda a la url */logout* symfony le deslogueará y le

redirigirá

a la url definida en *target,* en este caso le redirigirá a */*

El comando make:auth

En symfony 4.1 han añadido un comando nuevo **make:auth** que nos asiste en la configuración de la seguridad de nuestra aplicación.

Este comando es interactivo: nos va haciendo preguntas y al terminar nos genera el código y configuración necesarios.

> *php bin/console make:auth*

Previamente, conviene haber creado una entidad User bien manualmente o bien con el también nuevo comando **make:user**.

> *php bin/console make:user*

Enlace a la documentación oficial:
https://symfony.com/doc/current/security/form_login_setup.html