

Administración de usuarios en Symfony

Creación de un sistema de usuarios en Symfony junto con el componente de seguridad, añadir funcionalidades y emplear API keys

Contenido modificable

Si ves errores o quieres modificar/añadir contenidos, puedes [crear un pull request](#). Gracias



En esta sección se explican 5 temas importantes sobre la [administración de usuarios en una aplicación Symfony](#). Normalmente puede ser más conveniente emplear herramientas como [FOSUserBundle](#) o [Guard](#) que ya tienen funcionalidades incorporadas de las que aquí se explican, pero conviene conocer cómo funciona el sistema de usuarios para entender mejor las herramientas o para crear sistemas de usuarios más personalizados.

Índice de contenido

1. Cargar usuarios de la base de datos
2. Personalizar el formulario de login
3. Añadir la funcionalidad "Recordarme"
4. Cómo despertar a un usuario
5. Cómo autenticar a usuarios con API keys

1. Cargar usuarios de la base de datos

El sistema de seguridad de **Symfony** puede cargar usuarios desde cualquier lado: una base de datos, **Active Directory** o un **OAuth server**. En este caso se explica cómo se cargan usuarios de la base de datos con una **entidad Doctrine**.

Crear la entidad User

Suponemos que ya tenemos la **entidad User dentro de AppBundle** con los campos **id, username, password, email y isActive**.

```
// src/AppBundle/Entity/User.php
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\Entity\UserRepository")
 */
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;
        // puede no ser necesario, ver la sección salt debajo
        // $this->salt = md5(uniqid(null, true));
    }
}
```

```
}

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

public function getSalt()
{
    // podrías necesitar un verdadero salt dependiendo del encoder
    // ver la sección salt debajo
    return null;
}

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

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

public function eraseCredentials()
{
}

/** @see \Serializable::serialize() */
public function serialize()
{
    return serialize(array(
        $this->id,
        $this->username,
        $this->password,
        // ver la sección salt debajo
        // $this->salt,
    ));
}

/** @see \Serializable::unserialize() */
public function unserialize($serialized)
{
    list (
        $this->id,
        $this->username,
        $this->password,
        // ver la sección salt debajo
        // $this->salt
    ) = unserialize($serialized);
}
}
```

La clase entera es muy larga por lo que no se han mostrado algunos **getters** y **setters**, pero puedes generarlos con:

```
php bin/console doctrine:generate:entities AppBundle\Entity\User
```

Ahora nos aseguramos de actualizar los campos en la base de datos:

```
php bin/console doctrine:schema:update --force
```

De momento es una entidad normal, pero **para utilizar esta clase en el sistema de seguridad**, debe implementar **UserInterface**. Esto fuerza a la clase a tener los 5 métodos siguientes:

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

Al final de cada request, el objeto User se serializa en la sesión. En el siguiente **request**, se unserialize. Para ayudar a **PHP** a hacer esto correctamente se ha de emplear **Serializable**. Pero no se ha de serializar todo, sólo algunos campos (los anteriores que se han serializado y más si quieras si decides implementar **AdvancedUserInterface**). En cada request el **id** se utiliza para solicitar un **objeto User** fresco de la base de datos.

Configurar `security.yml` para cargar la entidad

Ahora que ya tenemos la entidad User que implementa **UserInterface**, sólo le tenemos que decírselo al **sistema de seguridad de Symfony**.

En este ejemplo crearemos un **formulario de login** (que se explica un poco más adelante). Symfony buscará una entidad User que coincida con el nombre de usuario y entonces comprobará el password:

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

    # ...

    providers:
        our_db_provider:
            entity:
                class: AppBundle:User
```

```

firewalls:
    main:
        anonymous: ~
        form_login:
            login_path: /login
            check_path: /login_check
        logout:
            path: /logout
            target: /
    # ...

```

Primero la sección **encoders** le dice a **Symfony** que los passwords en la base de datos se codificarán con **bcrypt**. Segundo, la sección **providers** crea un user provider llamado **_our_dbprovider** consultará a la entidad **AppBundle:User** por la propiedad **username**. El nombre **_our_dbprovider** no es importante, simplemente necesita coincidir con el valor de la key **provider** de tu firewall. Si no estableces la key **provider** en el firewall, se empleará el primer user provider.

Crear usuarios

Para añadir usuarios vamos a crear un **formulario de registro**. Es una entidad normal, así que no hay nada complejo, salvo que tenemos que **codificar el password**. Symfony proporciona un **service** para hacer esto automáticamente ('**_security.passwordencoder**').

Creamos el Form:

```

// src/AppBundle/Form/UserType.php
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Form\Extension\Core\Type\EmailType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\RepeatedType;
use Symfony\Component\Form\Extension\Core\Type>PasswordType;

class UserType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('email', EmailType::class)
            ->add('username', TextType::class)
            ->add('password', RepeatedType::class, array(
                'type' => PasswordType::class,
                'first_options' => array('label' => 'Password'),
                'second_options' => array('label' => 'Repetir Password'),
            ))
    }
}

```

```

    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults(array(
            'data_class' => 'AppBundle\Entity\User'
        ));
    }
}

```

Y un **controller** para arrancar el formulario. Es aquí donde se codifica el password y se emplea el service de **Symfony** para codificarlo:

```

// src/AppBundle/Controller/RegistrationController.php
namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use AppBundle\Form\UserType;
use AppBundle\Entity\User;
use Symfony\Component\HttpFoundation\Request;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class RegistrationController extends Controller
{
    /**
     * @Route("/register", name="user_registration")
     */
    public function registerAction(Request $request)
    {
        // 1) Construimos el formulario
        $user = new User();
        $form = $this->createForm(UserType::class, $user);

        // 2) Manejamos el envío (sólo pasará con POST)
        $form->handleRequest($request);
        if ($form->isSubmitted() && $form->isValid()) {
            // 3) Codificamos el password (también se puede hacer a través de un Doc
            $password = $this->get('security.password_encoder')
                ->encodePassword($user, $user->getPlainPassword());
            $user->setPassword($password);

            // 4) Guardar el User!
            $em = $this->getDoctrine()->getManager();
            $em->persist($user);
            $em->flush();

            // ... hacer cualquier otra cosa, como enviar un email, etc
            // establecer un mensaje "flash" de éxito para el usuario

            return $this->redirectToRoute('reemplazar_con_una_route');
        }
    }

    return $this->render(

```

```

        'registration/register.html.twig',
        array('form' => $form->createView())
    );
}
}

```

Insertamos en la template el formulario y ya podremos **registrar usuarios con el formulario**:

```

#{ app/Resources/views/registration/register.html.twig #}

{{ form_start(form) }}
{{ form_row(form.username) }}
{{ form_row(form.email) }}
{{ form_row(form.password.first) }}
{{ form_row(form.password.second) }}

<button type="submit">Registrarse!</button>
{{ form_end(form) }}

```

Como utilizamos el **algoritmo bcrypt** no hace falta salt, pero sino, sí que sería necesario. Todos los passwords han de hashearce con **salt**, pero **bcrypt** lo hace internamente. En este ejemplo empleamos bcrypt, por lo que **getSalt()** en la **clase User** devuelve **null**. Si utilizas un algoritmo diferente, necesitarás descomentar las líneas salt en la entidad User y añadir una **propiedad salt**.

Prohibir acceso a usuarios inactivos

Si la propiedad **isActive** de un usuario se establece a **false** (is_active es 0 en la base de datos), el usuario podrá igualmente logearse en el sitio normalmente. Esto se puede cambiar.

Para **excluir usuarios inactivos**, cambiamos la clase User para implementar **AdvancedUserInterface**. Esto extiende **UserInterface**, por lo que sólo necesitamos la nueva interface:

```

// 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 y unserialize deben ACTUALIZARSE
public function serialize()
{
    return serialize(array(
        // ...
        $this->isActive
    ));
}

public function unserialize($serialized)
{
    list (
        // ...
        $this->isActive
    ) = unserialize($serialized);
}
```

`AdvancedUserInterface` añade cuatro métodos extra para validar el estado de la cuenta:

- `isAccountNonExpired()`. Comprueba si la cuenta del usuario ha expirado.
 - `isAccountNonLocked()`. Comprueba si el usuario está bloqueado.
 - `isCredentialsNonExpired()`. Comprueba si las credenciales del usuario (password) han expirado.
 - `isEnabled()`. Comprueba si el usuario está activado.

Si cualquiera de estos devuelve `false`, el usuario no podrá logearse. Puedes elegir tener propiedades para estos también (en este ejemplo sólo `isActive` es una propiedad).

La diferencia entre los métodos es que cada uno devuelve un **error** ligeramente diferente, y éstos mensajes pueden traducirse cuando los renderizas en la template login para customizarlos.

Al utilizar `AdvancedUserInterface` también tenemos que añadir cualquiera de las propiedades utilizadas por estos métodos (como `isActive`) en los métodos `serialize`

y `unserialize`. Si no lo haces, el usuario podría no deserializarse correctamente de la sesión en cada request.

Ahora vamos a añadir el **formulario de login** muy básico. Ya tenemos configurado el firewall, vamos a crear el login controller:

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

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;

class LoginController extends Controller
{
    /**
     * @Route("/login", name="login_route")
     */
    public function loginAction(Request $request)
    {
        $authenticationUtils = $this->get('security.authentication_utils');

        // obtener el error de login si hay
        $error = $authenticationUtils->getLastAuthenticationError();

        // último nombre de usuario introducido por el usuario
        $lastUsername = $authenticationUtils->getLastUsername();

        return $this->render(
            'users/login.html.twig',
            array(
                // last username entered by the user
                'last_username' => $lastUsername,
                'error'           => $error,
            )
        );
    }

    /**
     * @Route("/login_check", name="login_check")
     */
    public function loginCheckAction()
    {
        // este controller no se ejecutará,
        // ya que la route se maneja por el sistema de seguridad
    }
}
```

Ahora ya podemos hacer **login**. Para hacer **logout**, ya tenemos configurada la `_securedarea` en el **firewall**, y simplemente tenemos que añadir la route en el archivo de routing:

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

Emplear una consulta personalizada para cargar al usuario

Vamos a hacer que el usuario pueda logearse empleando su username o su email, ya que ambos son únicos en la base de datos. El entity provider por defecto sólo puede consultar a través de una sola propiedad en el usuario.

Para hacer esto, vamos a hacer que **UserRepository** implemente **UserLoaderInterface**. Esta interface sólo requiere un método: **loadUserByUsername(\$username)**:

```
// src/AppBundle/Entity/UserRepository.php
namespace AppBundle\Entity;

use Symfony\Bridge\Doctrine\Security\User\UserLoaderInterface;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\Exception\UsernameNotFoundException;
use Doctrine\ORM\EntityRepository;

class UserRepository extends EntityRepository implements UserLoaderInterface
{
    public function loadUserByUsername($username)
    {
        $user = $this->createQueryBuilder('u')
            ->where('u.username = :username OR u.email = :email')
            ->setParameter('username', $username)
            ->setParameter('email', $username)
            ->getQuery()
            ->getOneOrNullResult();

        if (null === $user) {
            $message = sprintf(
                'Unable to find an active AppBundle:User object identified by "%s"',
                $username
            );
            throw new UsernameNotFoundException($message);
        }

        return $user;
    }
}
```

Ahora cuando un usuario entre, se llamará al método **loadUserByUsername()** en el **UserRepository**, y podrá introducir el nombre de usuario o el email.

Entendiendo serialize y cómo se guarda un usuario en la sesión

Una vez que el usuario hace **login**, el objeto **User** se serializa en la **sesión**. En el siguiente **request**, el objeto User se deserializa. Entonces, el valor de la propiedad **id** se usa para reconsultar un nuevo objeto User de la **base de datos**. Finalmente, el objeto User nuevo se compara con el usuario deserializado para asegurar que ambos representan al mismo usuario. Por ejemplo si el nombre de usuario en el segundo objeto User no coincide por cualquier razón, se cerrará la sesión del usuario por razones de seguridad.

Aunque esto ocurre de forma automática, hay algunos efectos secundarios.

Primero, la interface **Serializable** y sus métodos **serialize** y **unserialize** se han añadido para permitir que la clase User se serialize en la sesión. Esto puede ser necesario o no dependiendo de la instalación, pero probablemente es una buena idea. En teoría, sólo el **id** necesita ser serializado, porque el método **refreshUser()** refresca al usuario en cada request utilizando el **id**. Esto nos proporciona un objeto User "fresco".

Pero Symfony también utiliza el **username**, **salt** y **password** para verificar que el User no ha cambiado entre requests (también llama a los métodos **AdvancedUserInterface** si lo implementas). Fallar en la serialización de estas propiedades puede causar que se cierre la sesión en cada request. Si tu User implementa **EquatableInterface**, en lugar de comprobar estas propiedades llama al método **isEqualTo**, que comprueba las propiedades que quieras. Si no te interesa esta funcionalidad puedes omitirla.

2. Personalizar el formulario de login

Podemos **redireccionar al usuario cuando hace login** a donde especifiquemos con algunas opciones de configuración. Por defecto el formulario redireccionará a la URL que el usuario solicitó (la URL que mostró el formulario de login). Por ejemplo si el usuario solicita **ejemplo.com/posts**, después de que haga login se le enviará de vuelta a **ejemplo.com/posts**. Esto se hace guardando la URL solicitada en la sesión. Si no existe URL en la sesión (quizás el usuario accedió directamente a la página de login), entonces el usuario es redirigido a la página por defecto /. Este comportamiento se puede cambiar de varias formas:

1. Cambiar la página por defecto

La página por defecto puede establecerse (por ejemplo si la página previa a la que redirigir al usuario no está guardada en la sesión). Para esto modificamos la configuración:

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

```

main:
    form_login:
        # ...
        default_target_path: default_security_target

```

Ahora, si no existe URL en la sesión, el usuario será redirigido a la route `_default_securitytarget`.

2. Redirigir siempre a la página por defecto

Puedes hacer que los usuarios siempre sean redirigidos a la página por defecto independientemente de la URL que hayan solicitado anteriormente con la opción `_always_use_default_targetpath`:

```

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

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

```

3. Utilizar la URL de referencia

En caso de que no se haya guardado la URL en la sesión, puedes querer redireccionar utilizando el `HTTP_REFERER`, que normalmente será la misma. Se hace esto estableciendo la opción `_usereferer` a `true`:

```

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

    firewalls:
        main:
            # ...
            form_login:
                # ...
                use_referer: true

```

4. Controlar la redirección desde el formulario

Puedes sobreescibir cuando se redirige al usuario a través del formulario incluyendo un campo `hidden` con el nombre `_target_path`. Por ejemplo, para redirigir a la URL definida por una route `account`:

```

{# src/AppBundle/Resources/views/Security/login.html.twig #-}
{% if error %}
<div>{{ error.message }}</div>

```

```
{% endif %}

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

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

    <input type="hidden" name="_target_path" value="account" />

    <input type="submit" name="login" />
</form>
```

Ahora el usuario será redirigido al valor del campo hidden. El valor del atributo puede ser un **path relativo**, una **URL absoluta**, o un **nombre de route**. Puedes incluso cambiar el nombre del campo hidden del formulario cambiando la opción **_target_pathparameter**:

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

    firewalls:
        main:
            # ...
            form_login:
                target_path_parameter: redirect_url
```

5. Redireccionar cuando el usuario falle

Además de redireccionar al usuario después de un login existoso, puedes también establecer la URL a la que se redirigirá al usuario si el login falla (por ejemplo que se hayan introducido un nombre de usuario o password inválidos). Por defecto el usuario se redirige de vuelta al formulario de login. Puedes cambiar esto estableciendo la route **_loginfailure**:

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

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

3. Añadir la funcionalidad "Recordarme"

Una vez que el usuario está autenticado, sus credenciales se guardan en la sesión. Esto significa que cuando la sesión termina saldrán de la aplicación y tendrán que proporcionar los detalles de usuario de nuevo la próxima vez que quiera entrar. Puedes permitir que los usuarios permanezcan logeados por más tiempo que lo que dura la sesión utilizando una cookie con la opción del firewall `_rememberme`:

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

    firewalls:
        main:
            # ...
            remember_me:
                secret: '%secret%'
                lifetime: 604800 # 1 week in seconds
                path:   /
                # por defecto, esta funcionalidad está disponible en una
                # checkbox en el formulario de login, descomenta la
                # siguiente línea para activarlo siempre.
                #always_remember_me: true
```

El firewall `_rememberme` define las siguientes opciones de configuración:

- **secret**. Requerido. El valor empleado para encriptar el contenido de la cookie. Es frecuente emplear el valor definido en el archivo `app/config/parameters.yml`.
- **name**. Defecto: `REMEMBERME`. El nombre de la cookie para mantener al usuario logeado. Si activas la funcionalidad `remember_me` en varios firewalls de la misma aplicación, asegúrate de elegir diferentes nombres para las cookies de cada firewall, sino causará problemas.
- **lifetime**. Defecto: `315360000`. El número de segundos durante los que el usuario permanecerá logeado. Por defecto los usuarios están logeados por un año.
- **path**. Defecto: `/`. El path donde la cookie asociada se utiliza con esta funcionalidad. Por defecto la cookie se aplica al sitio entero pero puedes restringirlo a una sección específica (`/foro`, `/admin`, etc).
- **domain**. Defecto: `null`. El dominio donde la cookie asociada se utiliza con esta funcionalidad. Por defecto las cookies utilizan el dominio actual de `$_SERVER`.
- **secure**. Defecto: `false`. Si es `true`, la cookie con esta funcionalidad se envía a través de HTTPS.
- **httponly**. Defecto: `true`. Si es `true`, la cookie sólo es accesible a través de HTTP (por lo que no será accesible por lenguajes de scripting como JavaScript).
- **remember_me_parameter**. Defecto: `_remember_me`. El nombre del campo del formulario que decide si activar la funcionalidad "Recordarme".

- **always_remember_me**. Defecto: false. Si es true, el valor de `_remember_meparameter` se ignora y la funcionalidad "Recordarme" siempre se activa, independientemente de lo que quiera el usuario.
- **token_provider**. Defecto: null. Define el `service id` del token provider a utilizar. Por defecto, los `tokens` se guardan en una `cookie`. Por ejemplo, queremos guardar el token en una base de datos para no tener una versión hasheada del password en una cookie. El `DoctrineBridge` viene con `Symfony\Bridge\Doctrine\Security\RememberMe\DoctrineTokenProvider` que puedes utilizar.

Es una buena práctica dejar al usuario que elija si emplear la funcionalidad de recordarme o no, ya que no siempre puede ser apropiada. Poniendo como `name` de un `checkbox` `_rememberme` (o el nombre configurado en `_remember_meparameter`), la cookie se establecerá automáticamente cuando el checkbox esté seleccionado.

```
{# app/Resources/views/security/login.html.twig #}
{% if error %}
    <div>{{ error.message }}</div>
{% endif %}

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

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

    <input type="checkbox" id="remember_me" name="_remember_me" checked />
    <label for="remember_me">Keep me logged in</label>

    <input type="submit" name="login" />
</form>
```

El usuario estará logeado en visitas subsecuentes mientras la `cookie` permanezca válida.

Forzar al usuario a reautenticarse antes de acceder a ciertos resources

Cuando el usuario vuelve al sitio, está autenticado automáticamente por la información guardada en la cookie remember me. Esto permite al usuario acceder a resources protegidos como si el usuario se hubiera autenticado visitando el sitio.

En algunos casos puedes querer forzar al usuario a reautenticarse antes de acceder a ciertos resources. Por ejemplo puedes permitir a usuarios "recordarme" que vean cierta información, pero para modificarla tengan que autenticarse de nuevo.

El componente de seguridad proporciona una forma fácil de hacer esto. Además de los roles asignados a los usuarios, también se les da una de las siguientes etiquetas cuando se autentican:

- **IS_AUTHENTICATED_ANONYMOUSLY**. Automáticamente asignado a un usuario en una parte protegida por firewall pero que no se ha logeado todavía. Esto sólo es posible si se ha permitido el acceso anónimo.
- **IS_AUTHENTICATED_REMEMBERED**. Automáticamente asignado a un usuario autenticado con una cookie remember me.
- **IS_AUTHENTICATED_FULLY**. Automáticamente asignado a un usuario que ha proporcionado los detalles de login en la sesión actual.

Tienen orden jerárquico, de forma que **IS_AUTHENTICATED_FULLY** engloba a las otras dos, y **IS_AUTHENTICATED_REMEMBERED** incluye **IS_AUTHENTICATED_ANONYMOUSLY**.

Puedes utilizar estas keywords para controlar el acceso además de los roles asignados. Por ejemplo quieres que un usuario pueda ver su cuenta en `/account` cuando esté autenticado por cookie pero tiene que proporcionar sus credenciales para editarla.

```
// ...
use Symfony\Component\Security\Core\Exception\AccessDeniedException

// ...
public function editAction()
{
    $this->denyAccessUnlessGranted('IS_AUTHENTICATED_FULLY');

    // ...
}
```

También se puede hacer mediante **anotaciones**:

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

/**
 * @Security("has_role('IS_AUTHENTICATED_FULLY')")
 */
public function editAction($name)
{
    // ...
}
```

Si tenemos **control de acceso** en la **configuración de seguridad** que requiere que el usuario tenga el role **ROLE_USER** para acceder a cualquier área de la cuenta, tendremos la siguiente situación:

- Si un usuario no autenticado (o autenticado de forma anónima) intenta acceder al área /account, se le pedirá que se autentique.
- Una vez que el usuario ha introducido su usuario y contraseña, asumiendo que el usuario recibe el role **ROLE_USER**, el usuario tendrá **IS_AUTHENTICATED_FULLY** y podrá acceder a cualquier página en la sección /account, incluído el controller editAction.
- Si finaliza la sesión del usuario, cuando el usuario vuelva al sitio, podrá acceder a cada página de /account excepto la de editar, sin ser forzado a reautenticarse. Sin embargo, cuando intente acceder al controller editAction, se forzará al usuario a que se autentique de nuevo.

4. Cómo despersonalizar un usuario

A veces puede ser útil poder cambiar de un usuario a otro sin tener que salir y entrar de nuevo (por ejemplo para hacer debugging o intentar entender un bug que un usuario ve pero no puedes reproducir). Esto se puede hacer fácilmente activando el listener del firewall `_switchUser`:

```
# app/config/security.yml
security:
    # ...
    firewalls:
        main:
            # ...
            switch_user: true
```

Para cambiar a otro usuario, simplemente añade un **query string** al parámetro `_switchUser` y el nombre de usuario como el valor de la URL actual:

`http://example.com/somewhere?_switch_user=thomas`

Para volver al usuario original, emplea el nombre de usuario especial `_exit`:

`http://example.com/somewhere?_switch_user=_exit`

Durante la despersonalización, se proporciona al usuario un role especial llamado `ROLE_PREVIOUS_ADMIN`. En la template, por ejemplo, este role puede usarse para mostrar un link y salir de la despersonalificación:

```
{% if is_granted('ROLE_PREVIOUS_ADMIN') %}
    <a href="{{ path('homepage', {'_switch_user': '_exit'}) }}>Salir de la despersonalización</a>
{% endif %}
```

En algunos casos puedes necesitar obtener el objeto que representa el usuario despersonalizado en lugar del usuario despersonalizado. Con el siguiente snippet podemos iterar en los roles del usuario hasta que encuentre uno que tenga un objeto **SwitchUserRole**:

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

$authChecker = $this->get('security.authorization_checker');
$tokenStorage = $this->get('security.token_storage');

if ($authChecker->isGranted('ROLE_PREVIOUS_ADMIN')) {
    foreach ($tokenStorage->getToken()->getRoles() as $role) {
        if ($role instanceof SwitchUserRole) {
            $impersonatingUser = $role->getSource()->getUser();
            break;
        }
    }
}
```

Esta funcionalidad sólo se ha de proporcionar a un grupo pequeño de usuarios. Por defecto, el acceso está restringido a usuarios que tienen el role **ROLE_ALLOWED_TO_SWITCH**. El nombre de este role puede modificarse con el ajuste **role**. Para seguridad extra, puedes también cambiar el parámetro del query con el ajuste **parameter**:

```
# app/config/security.yml
security:
    # ...
    firewalls:
        main:
            # ...
            switch_user: { role: ROLE_ADMIN, parameter: _want_to_be_this_user }
```

Eventos

El firewall lanza el evento `_security.switchuser` justo después de que la despersonalización se completa. El **SwitchUserEvent** se pasa al listener y puedes usarlo para obtener al usuario que estás despersonalizando.

Con lo siguiente podremos **cambiar el locale**. Primero la configuración:

```
# app/config/services.yml
services:
    app.switch_user_listener:
        class: AppBundle\EventListener\SwitchUserListener
        tags:
            - { name: kernel.event_listener, event: security.switch_user, method: onS
```

La implementación del listener supone que tu entidad User tiene un método `getLocale()`:

```
// src/AppBundle/EventListener/SwitchUserListener.php
namespace AppBundle\EventListener;

use Symfony\Component\Security\Http\Event\SwitchUserEvent;

class SwitchUserListener
{
    public function onSwitchUser(SwitchUserEvent $event)
    {
        $event->getRequest()->getSession()->set(
            '_locale',
            $event->getTargetUser()->getLocale()
        );
    }
}
```

5. Cómo autenticar a usuarios con API keys

Para conseguir funcionalidades como esta es mejor emplear el sistema de autenticación con Guard, pero conviene entender cómo se hace.

Actualmente es muy frecuente **autenticar a un usuario a través de una API key** (cuando se desarrolla un web service por ejemplo). El API key se proporciona en cada request y se pasa como parámetro query string o a través del header HTTP.

El API key Authenticator

Autenticar a un usuario según la información del Request debería hacerse a través de un mecanismo de preautenticación. `SimplePreAuthenticatorInterface` te permite implementar el esquema fácilmente.

En este ejemplo, un token se lee desde un parámetro query `apikey`, el nombre de usuario adecuado se carga desde ese valor y entonces se crea el objeto User:

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

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Security\Core\Authentication\Token\PreAuthenticatedToken;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Exception\AuthenticationException;
use Symfony\Component\Security\Core\Exception\CustomUserMessageAuthenticationException;
use Symfony\Component\Security\Core\Exception\BadCredentialsException;
use Symfony\Component\Security\Core\User\UserProviderInterface;
use Symfony\Component\Security\Http\Authentication\SimplePreAuthenticatorInterface;

class ApiKeyAuthenticator implements SimplePreAuthenticatorInterface
```

```

{
    public function createToken(Request $request, $providerKey)
    {
        // look for an apikey query parameter
        $apiKey = $request->query->get('apikey');

        // or if you want to use an "apikey" header, then do something like this:
        // $apiKey = $request->headers->get('apikey');

        if (!$apiKey) {
            throw new BadCredentialsException('No API key found');

            // or to just skip api key authentication
            // return null;
        }

        return new PreAuthenticatedToken(
            'anon.',
            $apiKey,
            $providerKey
        );
    }

    public function authenticateToken(TokenInterface $token, UserProviderInterface $u
    {
        if (!$userProvider instanceof ApiKeyUserProvider) {
            throw new \InvalidArgumentException(
                sprintf(
                    'The user provider must be an instance of ApiKeyUserProvider (%s
                    get_class($userProvider)
                )
            );
        }

        $apiKey = $token->getCredentials();
        $username = $userProvider->getUsernameForApiKey($apiKey);

        if (!$username) {
            // CAUTION: this message will be returned to the client
            // (so don't put any un-trusted messages / error strings here)
            throw new CustomUserMessageAuthenticationException(
                sprintf('API Key "%s" does not exist.', $apiKey)
            );
        }

        $user = $userProvider->loadUserByUsername($username);

        return new PreAuthenticatedToken(
            $user,
            $apiKey,
            $providerKey,
            $user->getRoles()
        );
    }
}

```

```

public function supportsToken(TokenInterface $token, $providerKey)
{
    return $token instanceof PreAuthenticatedToken && $token->getProviderKey() ==
}
}

```

Una vez que lo hayamos configurado (vemos más adelante como hacerlo), podremos autenticarnos añadiendo el parámetro `apikey` al query string:
<http://example.com/admin/foo?apikey=37b51d194a7513e45b56f6524f2d51f2>.

El proceso de autenticación tiene varios pasos, y tu implementación puede diferir:

1. **createToken**. Al principio del ciclo request, Symfony llama a `createToken()`. Tu tarea es crear un objeto token que contiene toda la información del request que necesitas para autenticar al usuario (el parámetro query `apikey`). Si no se proporciona esa información, se lanzará la excepción `BadCredentialsException`. Puedes también devolver `null` y saltar la autenticación para que Symfony intente otro método de autenticación si existe. En caso de devolver `null` asegúrate de activar `anonymous` en tu firewall, así podrás obtener un `AnonymousToken`.
2. **supportsToken**. Después de que Symfony llame a `createToken()`, llamará a `supportsToken()` (o cualquier otro listener de autenticación) para averiguar quién manejará el token. Esto es una forma de permitir varios mecanismos de autenticación para el mismo firewall (así puedes por ejemplo intentar primero autenticar al usuario con un certificado o una API key y volver a un formulario de login).
3. **authenticateToken**. Si `supportsToken()` devuelve true, Symfony llamará a `authenticateToken()`. La clase externa `$userProvider` permite cargar información del usuario. Lo vemos más adelante.

En este ejemplo ocurren varias cosas en `authenticateToken()`:

1. Primero empleamos `$userProvider` para buscar al `$username` que corresponda al `$apiKey`.
2. Segundo usamos el `$userProvider` de nuevo para cargar o crear un objeto `User` para el `$username`.
3. Y tercero creamos un `authenticated token` (un token con al menos un `role`) que tiene los roles adecuados y el objeto `User` ligado. El objetivo es usar el `$apiKey` para encontrar o crear un objeto `User`. Cómo hacer esto (por ejemplo consultar una base de datos) y la clase exacta de tu objeto `User` puede variar. Estas diferencias serán más evidentes en el `user provider`.

El User Provider

El `$userProvider` puede ser cualquier user provider. En este ejemplo el `$apiKey` se emplea para encontrar el nombre de usuario del usuario. Esto se hace con el método `getUsernameForApiKey()`, que creamos para este caso.

El **user provider** puede ser algo así:

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

use Symfony\Component\Security\Core\User\UserProviderInterface;
use Symfony\Component\Security\Core\User\User;
use Symfony\Component\Security\Core\User\UserInterface;
use Symfony\Component\Security\Core\Exception\UnsupportedUserException;

class ApiKeyUserProvider implements UserProviderInterface
{
    public function getUsernameForApiKey($apiKey)
    {
        // Busca el username según el token de la base de datos, a través
        // de una API call, o haz algo distinto
        $username = ...;

        return $username;
    }

    public function loadUserByUsername($username)
    {
        return new User(
            $username,
            null,
            // los roles del usuario - puedes elegir determinarlos
            // dinámicamente de alguna forma basándote en el usuario
            array('ROLE_USER')
        );
    }

    public function refreshUser(UserInterface $user)
    {
        // Esto se emplea para guardar la autenticación en la sesión,
        // pero en este ejemplo el token se envía en cada request,
        // por lo que la autenticación puede ser stateless. Lanzando esta excepción
        // es adecuado para hacerlo stateless
        throw new UnsupportedUserException();
    }

    public function supportsClass($class)
    {
        return 'Symfony\Component\Security\Core\User\User' === $class;
    }
}
```

Ahora registra el **user provider como service**:

```
# app/config/services.yml
services:
    api_key_user_provider:
        class: AppBundle\Security\ApiKeyUserProvider
```

La lógica de `getUsernameForApiKey()` depende de ti. Puedes de alguna forma transformar el API key (por ejemplo 43b31h) en un nombre de usuario (ejemplo johndoe) buscando información en una tabla token de la base de datos.

Lo mismo para `loadUserByUsername()`. En este ejemplo, la clase User del core de **Symfony** se crea. Tiene sentido si no necesitas guardar ninguna información extra en tu objeto User. Pero si tienes que guardar más información, puedes crear tu propia clase User con más propiedades.

Finalmente asegúrate de que `supportsClass()` devuelve true para objetos User con la misma clase sea cual sea el usuario que devuelve `loadUserByUsername()`. Si tu autenticación es stateless como en este ejemplo (por ejemplo esperas que el usuario envíe el API key con cada request de forma que no tengas que guardar el login en la sesión), entonces puedes simplemente lanzar la excepción `UnsupportedUserException` en `refreshUser()`.

Administrar errores en la autenticación

Para que `ApiKeyAuthenticator` muestre correctamente un estado HTTP 403 con credenciales incorrectas o la autenticación falla, necesitarás implementar `AuthenticationFailureHandlerInterface` en tu `Authenticator`. Esto proporcionará un método `onAuthenticationFailure` que puedes usar para crear una respuesta de error:

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

use Symfony\Component\Security\Core\Exception\AuthenticationException;
use Symfony\Component\Security\Http\Authentication\AuthenticationFailureInterface;
use Symfony\Component\Security\Http\Authentication\SimplePreAuthenticatorInterface;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Request;

class ApiKeyAuthenticator implements SimplePreAuthenticatorInterface, AuthenticationFailureInterface
{
    // ...

    public function onAuthenticationFailure(Request $request, AuthenticationException $exception)
    {
        return new Response(
            // Esto contiene información acerca de por qué falló la autenticación
            // úsala, o devuelve tu propio mensaje
            strtr($exception->getMessageKey(), $exception->getMessageData()),
            403
        );
    }
}
```

```

    }
}

```

Configuración

Una vez que tenemos preparada la **ApiKeyAuthenticator**, tenemos que registrarla como service y usarla en la configuración de seguridad.

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

    apikey_authenticator:
        class: AppBundle\Security\ApiKeyAuthenticator
        public: false
```

Ahora la activamos junto con el user provider con las keys **_simplepreauth** y **provider**:

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

    firewalls:
        secured_area:
            pattern: ^/admin
            stateless: true
            simple_preauth:
                authenticator: apikey_authenticator
                provider: api_key_user_provider

    providers:
        api_key_user_provider:
            id: api_key_user_provider
```

Eso es todo, ahora se llamará a **ApiKeyAuthenticator** al principio de cada request y se iniciará el proceso de autenticación.

El parámetro de configuración **stateless** previene de que **Symfony** intente guardar información de autenticación en la sesión, ya que no es necesario porque el cliente enviará el **apikey** en cada request.

Guardar la autenticación en la sesión

Hasta ahora se ha descrito una situación en la que el token de autenticación se envía en cada request. Pero en algunas situaciones (como en OAuth) el token puede enviarse en sólo un request. En este caso tendremos que autenticar al usuario y guardar la autenticación en la sesión de forma que el usuario esté logeado en cada request subsecuente.

Para que esto funcione, quitamos el parámetro `stateless` de la configuración del firewall o lo ponemos en `false`.

Incluso cuando el token se guarda en la sesión, las credenciales, en este caso la API key (`$token->getCredentials()`) no se guardan en la sesión por razones de seguridad. Para aprovechar la sesión, actualiza `ApiKeyAuthenticator` para ver si el token guardado tiene un objeto User válido que pueda usarse:

```
// src/AppBundle/Security/ApiKeyAuthenticator.php

// ...
class ApiKeyAuthenticator implements SimplePreAuthenticatorInterface
{
    // ...

    public function authenticateToken(TokenInterface $token, UserProviderInterface $userProvider)
    {
        if (!$userProvider instanceof ApiKeyUserProvider) {
            throw new \InvalidArgumentException(
                sprintf(
                    'El user provider debe ser una instancia de ApiKeyUserProvider (%s)',
                    get_class($userProvider)
                )
            );
        }

        $apiKey = $token->getCredentials();
        $username = $userProvider->getUsernameForApiKey($apiKey);

        // User es la entidad que representa tu usuario
        $user = $token->getUser();
        if ($user instanceof User) {
            return new PreAuthenticatedToken(
                $user,
                $apiKey,
                $providerKey,
                $user->getRoles()
            );
        }

        if (!$username) {
            // este mensaje se devolverá al cliente
            throw new CustomUserMessageAuthenticationException(
                sprintf('API Key "%s" does not exist.', $apiKey)
            );
        }

        $user = $userProvider->loadUserByUsername($username);

        return new PreAuthenticatedToken(
            $user,
            $apiKey,
            $providerKey,
            $user->getRoles()
        );
    }
}
```

```

    );
}
// ...
}

```

Guardar información de autenticación en la sesión funciona de la siguiente forma:

1. Al final de cada request, **Symfony** serializa el objeto token (devuelto por `authenticateToken()`) que también serializa el objeto User (ya que está establecido en una propiedad en el token).
2. En el siguiente request el token se deserializa y el objeto User se pasa a la función `refreshUser()` del user provider.

El segundo paso es el importante, **Symfony** llama a `refreshUser()` y te pasa el objeto user que se serializó en la sesión. Si tus usuarios están guardados en la base de datos, podrías reconsultar una versión fresca del usuario para asegurarte que no está obsoleto. Pero independientemente de los requisitos, `refreshUser()` debería devolver el objeto User:

```

// src/AppBundle/Security/ApiKeyUserProvider.php

// ...
class ApiKeyUserProvider implements UserProviderInterface
{
    // ...

    public function refreshUser(UserInterface $user)
    {
        // $user es el objeto User que estableces en el token dentro de authenticateT
        // después de que se haya deserializado de la sesión

        // puedes usar $user para consultar la base de datos para un usuario fresco
        // $id = $user->getId();
        // usa $id para la consulta

        // no lo obtienes los datos de una base de datos y simplemente creas un
        // objeto User (como en este ejemplo), puedes simplemente devolverlo
        return $user;
    }
}

```

También querrás asegurarte de que tu objeto User se serializa correctamente. Si tu objeto User tiene propiedades privadas, PHP no puede serializarlas. En ese caso podrías obtener un objeto User que tiene un valor null para cada propiedad.

Sólo autenticar para ciertas URLs

Hasta ahora hemos supuesto que quieras buscar la autenticación `apikey` en cada request. Pero en algunas situaciones (como en OAuth), sólo necesitas buscar **información de autenticación** una vez que el usuario ha alcanzado cierta URL (la URL de redirección en OAuth).

Simplemente tenemos que ver la URL actual antes de crear el token en `createToken()`:

```
// src/AppBundle/Security/ApiKeyAuthenticator.php

// ...
use Symfony\Component\Security\Http\HttpUtils;
use Symfony\Component\HttpFoundation\Request;

class ApiKeyAuthenticator implements SimplePreAuthenticatorInterface
{
    protected $httpUtils;

    public function __construct(HttpUtils $httpUtils)
    {
        $this->httpUtils = $httpUtils;
    }

    public function createToken(Request $request, $providerKey)
    {
        // establece la única URL donde deberíamos buscar información auth
        // y sólo devuelve el token si estamos en esa URL
        $targetUrl = '/login/check';
        if (!$this->httpUtils->checkRequestPath($request, $targetUrl)) {
            return;
        }

        // ...
    }
}
```

Esto emplea la clase `HttpUtils` para comprobar si la URL actual coincide con la URL que buscas. En este caso, la URL (`/login/check`) se ha puesto directamente en la clase, pero puedes inyectarla también como segundo argumento constructor.

Ahora simplemente actualiza la configuración del service:

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

    apikey_authenticator:
        class: AppBundle\Security\ApiKeyAuthenticator
        arguments: ["@security.http_utils"]
        public: false
```



Copyright © Diego Lázaro 2018

Sitio construido con [Symfony](#) & [Semantic-UI](#)