Le composant sécurité de Symfony

Notes

Ce document se veut être un guide pour tout débutant sur le framework Symfony, voulant en apprendre plus sur le composant sécurité. À la fin de ce document, vous aurez les connaissances suffisante pour mettre en place un système d'authentification pour votre application.
Ce guide s'inspire de la documentation Symfony.

Mécanismes de la configuration

Paramétrage initial

La sécurité de Symfony est un système très complet, et son configuration peut, au premier abord, paraître déroutante. Dans ce document, nous allons nous concentrer sur un nombre limité de paramètres permettant de couvrir la quasi-totalité des besoins en sécurisation d'une application. L'ensemble des paramètres de sécurité est disponible à l'URL suivante :
https://symfony.com/doc/current/reference/configuration/security.html

La sécurité est entièrement configurable depuis le fichier `app/config/security.yaml`. Initialement, voici ce que l'on peut y trouver :

```
security:
      providers:
            in_memory:
                  memory: ~
      firewalls:
            dev:
                  pattern: ^/(_(profiler|wdt)|css|images|js)/
                  security: false
            main:
                  anonymous: ~
```

Il y a à ce stade 2 clés de configuration : `providers` et `firewalls`. Nous reviendrons plus tard sur `providers`, définissant les utilisateurs pouvant s'identifier.
La clé `firewalls` est sans aucun doute l'élément le plus important : elle définit les URLs demandant (ou non) une authentification, ainsi que la façon dont il s'agit de s'authentifier. Comme son nom l'indique, elle s'apparente au pare-feu de l'application. C'est typiquement la première chose à configurer.

Il y a par défaut 2 clés directement sous `firewalls`, qui définissent donc 2 pare-feux (leurs noms n'a strictement aucune importance, et sont arbitraires) :

- `dev`, s'assurant que les URLs utilisés par les outils de développement sur Symfony, telles que `/_profiler` ou `/_wdt`, ne sont pas sécurisées.
- `main`, représentant toutes les autres URLs, utilisées par l'application.

Sous chacun de ces pare-feux, nous observons dors et déjà 3 clés différentes servant à définir son comportement :

- `pattern`, définissant, via une expression *regex*, les URLs affectées au pare-feu. Si cette clé n'a aucune valeur, alors le pare-feu affecte toutes les URLs, comme c'est le cas pour le `main` ci-dessus.
- `security`, un booléen indiquant si le pare-feu demande une authentification pour les URLs définies (`true` par défaut).
- `anonymous` qui, si elle est présente, authentifie tout client accédant aux URLs du pare-feu en tant que `anon.`.

Initialement, vos URLs sont donc sous le pare-feu `main`, et vous êtes authentifié en tant qu'utilisateur `anon.`.
Le composant sécurité de Symfony comprend en fait 2 systèmes : l'authentification, communément appelé "pare-feu", et l'autorisation, appelé "contrôle d'accès". Une fois le pare-feu "traversé" via une authentification, le contrôle d'accès prend le relais : il vérifie si le client authentifié a bien le droit d'accéder à l'URL en question. Si le client est authentifié en tant que `anon.`, et que l'URL demandée nécessite un droit spécifique, une authentification dite "complète" lui sera demandé : le contrôle d'accès indiquera au pare-feu qu'il est nécessaire que le client s'authentifie, et il devra s'identifier comme un des utilisateurs définie par `providers`. Si le client est déjà complètement authentifié et qu'il n'a pas le droit d'accéder à l'URL, l'accès lui sera refusé.
Nous verrons par la suite comment fonctionne en détail le contrôle d'accès, permettant d'attribuer des rôles aux utilisateurs. Cependant, on se rend déjà compte de l'utilité d'être connecté en tant que `anon.` : le pare-feu authentifie automatiquement le client, le "laisse passer", puis le contrôle d'accès vérifie s'il a le droit d'accéder à l'URL. Si un droit est explicitement attribué à cette URL, le pare-feu reprend le relais, en demandant une authentification complète.

Première demande d'authentification

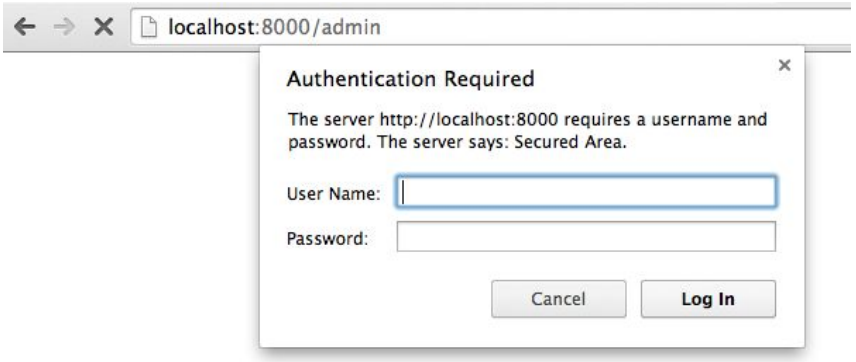
Nous pouvons de manière très simple effectuer un premier test du système. Ajoutons au pare-feu main la clé http_basic afin de préciser que l'on veut s'authentifier via le *HTTP authentication framework* (RFC 7235) :

```
security:
# ...
firewalls:
      # ...
      main:
            anonymous: ~
            http_basic: ~
```

Créez par exemple une route associé à l'URL `/admin`. Cette URL sera sous le pare-feu `main`, et vous pouvez y accéder : vous êtes connecté en tant que `anon.`, et il n'y a pas de contrainte quant au contrôle d'accès.
Maintenant, indiquons au contrôle d'accès que `/admin` requiert que l'utilisateur ait un rôle (un droit) spécifique :

```
security:
      # ...
      firewalls:
            # ...
            main:
                  # ...
      access_control:
            - { path: ^/admin, roles: ROLE_ADMIN }
```

L'expression *regex* associée à l'attribut `path` définie l'ensemble des URLs concernées par le contrôle d'accès. L'utilisateur authentifié doit disposer des rôles renseignés dans l'attribut `roles` pour pouvoir accéder aux URLs.
Essayez maintenant de vous rendre sur l'URL `/admin` : le contrôle d'accès s'est aperçu que `anon.` ne dispose pas du rôle `ROLE_ADMIN`, requis pour accéder à `/admin`. Le pare-feu amorce alors une demande d'authentification au client : un prompt apparaît sur le navigateur. Nous verrons plus tard comment intégrer un formulaire de connexion.



Autorisation

Le contrôle d'accès vérifie si l'utilisateur courant dispose du ou des droits nécessaires pour accéder à l'URL demandée : il est chargé d'autoriser ou de refuser l'accès. Dans une application Symfony, ces droits permettant de décider qui peut accéder à quoi sont modélisés par des rôles, et tout utilisateur en a au moins un.
Comme vu dans la section précédente, le ou les rôles nécessaires pour accéder à une URL donnée sont définis sous la clé `access_control` :

```
security:
      # ...
      firewalls:
            # ...
            main:
                  # ...
      access_control:
                  - { path: ^/admin, roles: ROLE_ADMIN }
```

Un format tableau peut aussi être attribué à la clé `roles`, ce qui est utile dans le cas où l'accès à une URL nécessite plusieurs rôles.
La ligne ajoutée sur la clé `access_control` stipule que l'utilisateur courant doit disposer du rôle `ROLE_ADMIN` pour pouvoir accéder aux URLs commençant par `/admin`. Le nom des rôles affectés aux URLs est totalement arbitraire. Cependant, la convention utilisée est la suivante : un rôle doit être en majuscule, et commencer par `ROLE_`.
Deux rôles sont prédéfinis par Symfony : `IS_AUTHENTICATED_ANONYMOUSLY` et `IS_AUTHENTICATED_FULLY`. Ils peuvent être utilisés par le contrôle d'accès afin de déterminer si l'utilisateur est authentifié anonymement ou s'il a été chargé depuis la fournisseur d'utilisateur.

Symfony met en place une gestion de la hiérarchie des rôles : cela peut éviter d'avoir à en associer plusieurs à un utilisateur. La hiérarchie se définie sous la clé `role_hierarchy`, immédiatement sous `security` :

```
security:
       # ...
       role_hierarchy:
              ROLE_ADMIN: ROLE_USER
              ROLE_SUPER_ADMIN: [ROLE_ADMIN, ROLE_ALLOWED_TO_SWITCH]
```

Avec cette configuration, tout utilisateur ayant le rôle ROLE_ADMIN dispose également du rôle ROLE_USER. Un tableau peut être utiliser pour attribuer plusieurs rôles subordonnés.

Configuration du fournisseur d'utilisateur

Au moment où l'on entre nos identifiants pour nous connecter, Symfony doit charger les informations de l'utilisateur. Cela se fait grâce au fournisseur d'utilisateur, définissant la façon dont les utilisateurs sont chargés à la connexion.
Les fournisseurs d'utilisateurs sont renseignés sous la clé `providers`.
Nous pouvons, afin de nous familiariser avec ce concept, définir de manière très simple un fournisseur d'utilisateur en écrivant "en dur", dans le fichier `app/config/security.yaml` :

```
security:
      providers:
            in_memory:
                  memory:
                        users:
                              ryan:
                                    password: ryanpass
                                    roles: 'ROLE_USER'
                              admin:
                                    password: adminpass
                                    roles: 'ROLE_ADMIN'
      # ...
```

Les fournisseurs sous la clé `in_memory` définissent leurs utilisateurs au sein même du fichier de configuration. Ici, nous afin un fournisseur dit "en mémoire", nommé `memory`. Directement sous la clé `users` sont les noms d'utilisateurs avec lesquels il s'agit de s'authentifier. Lorsqu'un client se connecte en tant qu'utilisateur issu d'un fournisseur en mémoire, il est modélisé par la classe `Symfony\Component\Security\Core\User\User`.
Une fois un fournisseur d'utilisateur défini, il faut indiquer l'encodeur de mot de passe qui sera utilisé pour vérifier le mot de passe entré par le client. Dans notre exemple, nos utilisateurs `ryan` et `admin` ont des mots de passe non-encodés (`ryanpass` et `adminpass`). Nous informons donc à Symfony qu'à la comparaison des mots de passe d'utilisateurs modélisé par la classe `Symfony\Component\Security\Core\User\User`, nous ne voulons pas encoder le mot de passe entré par le client :

```
security:
      # ...
      encoders:
            Symfony\Component\Security\Core\User\User: plaintext
      # ...
```

Nous pouvons désormais nous connecter en tant qu'un des deux utilisateurs défini dans le fournisseur memory, en renseignant `ryan` ou `admin` comme nom d'utilisateur, et `ryanpass` ou `adminpass` comme mot de passe.

Considérons maintenant le chargement de l'utilisateur depuis une base de données, grâce à l'ORM utilisé par Symfony (Doctrine dans la très grande majorité des cas). Nous verrons dans la prochaine section comment implémenter en détail l'entité "utilisateur" ; il est cependant à noter qu'une telle entité se doit d'implémenter l'interface `UserInterface` afin de pouvoir être exploitée par le système d'authentification de Symfony.
Le fournisseur d'utilisateurs, pour un chargement depuis une base de données, se configure de la façon suivante :

```
security:
      encoders:
            AppBundle\Entity\User:
                  algorithm: bcrypt
      # ...
      providers:
            our_db_provider:
                  entity:
                        class: AppBundle\Entity\User
                        property: username
      # ...
```

Sous la clé `providers`, nous avons créé le fournisseur d'utilisateur `our_db_provider`. La clé `entity` permet d'indiquer deux choses :

- `class` : l'entité cherchée par l'ORM à l'authentification
- `property` : la propriété correspondant au nom d'utilisateur (saisi avec le mot de passe lors de l'authentification)

Sous la clé `encoders`, nous indiquons l'algorithme avec lequel les mots de passe sont cryptés en base de donnée (dans la table correspondant à l'entité utilisateur). Ici, il s'agit de `bcrypt`.

Implémentation de l'utilisateur

Utilisation de l'interface UserInterface

Reprenons l'exemple précédent, où il s'agit d'utiliser une table `app_user` définie par l'entité `AppBundle\Entity\User`, avec un nom d'utilisateur représenté par la propriété `username`, et un mot de passe encodé à l'aide de l'algorithme `bcrypt`.
Afin de pouvoir être exploitée par le composant sécurité, notre classe doit implémenter l'interface `UserInterface` :

```
namespace AppBundle\Entity;

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

/**
* @ORM\Table(name="app_user")
* @ORM\Entity(repositoryClass="AppBundle\Repository\UserRepository")
*/
class User implements UserInterface
{
      # ...

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

      public function getSalt()
      {
            return null;
      }

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

      public function getRoles()
      {
            return ['ROLE_USER'];
      }

      public function eraseCredentials()
      {
      }
}
```

Notre utilisateur, en base de données, est persisté comme suit :

```
mysql> SELECT * FROM app_user;
+----+----------+--------------------------------------+--------------------+
| id | username | password                             | email              |
+----+----------+--------------------------------------+--------------------+
|  1 | admin    | $2a$08$jHZj/wJfcVKlIwr5AvR78euJxYK7KuC | admin@example.com  |
+----+----------+--------------------------------------+--------------------+
```

Il y a 4 méthodes à implémenter au sein de l'entité `User` :

- `getUsername` : retourne la propriété représentant le nom d'utilisateur unique.
- `getSalt` : retourne le nom de l'algorithme à utiliser pour encoder le mot de passe avant de persister l'instance d'utilisateur. Si l'algorithme est `bcrypt` ou `argon2i`, il peut être renseigné sous le clé `encoders` dans le fichier de configuration `security.yml`, et `getSalt` doit retourner `null`.
- `getPassword` : retourne le mot de passe.
- `getRoles` : retourne les rôles de l'utilisateur, permettant de connaître ses permissions
- `eraseCredentials` : simplement utilisée pour supprimer les possibles mots de passe stockés en clair dans une propriété non-persistée de l'instance.

Les sections qui suivent vont proposer l'implémentation de mécanismes d'inscription et de connexion. Cependant, en persistant un utilisateur en base de données et en encodant manuellement son mot de passe, nous pouvons déjà réaliser une authentification complète.

Inscription

Voyons maintenant un mécanisme permettant de s'inscrire sur notre application.

La première étape est d'affecter une URL à une page d'inscription : utilisons `/register`. Nous devons nous assurer que le contrôle d'accès autorise tout utilisateur anonyme à accéder à cette URL.

```
security:
      # ...
      access_control:
            - { path: ^/, roles: ROLE_USER }
            - { path: ^/register, roles: IS_AUTHENTICATED_ANONYMOUSLY }
```

Nous allons utiliser une entité `AppBundle\Entity\User` ayant simplement 3 propriétés mappées avec l'ORM :

- `id: integer`
- `username: string`
- `password: string`

Nous ajoutons une 4ème propriété, non-mappée : `plainPassword`, permettant de stocker temporairement le mot de passe en clair à l'inscription, avant d'affecter sa version encodée à la propriété `password`.

```
# ...
class User implements UserInterface
{
      /**
      * @ORM\Column(name="id", type="integer")
      * @ORM\Id
      * @ORM\GeneratedValue(strategy="AUTO")
      */
      private $id;

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

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

      private $plainPassword;

      # ...
}
```

Les *getters* et *setters* sont omis ici, pour plus de clarté. Les méthodes de `UserInterface` seront implémentées de la façon suivante :

```
# ...
class User implements UserInterface
{
	# ...

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

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

	public function getSalt()
	{
		return null;
	}

	public function eraseCredentials()
	{
		$this->plainPassword = null;
	}

	# ...
}
```

Pour rappel, voici la façon dont nous allons configurer le fournisseur d'accès :

```
security:
	encoders:
		AppBundle\Entity\User:
			algorithm: bcrypt
	# ...
	providers:
		our_db_provider:
			entity:
				class: AppBundle\Entity\User
				property: username
	# ...
```

Maintenant, créons une classe héritant de `AbstractType`, représentant un formulaire d'inscription. Il va permettre d'hydrater une instance de `User` à l'inscription, que nous persisterons en base de donnée à la soumission du formulaire valide :

```
namespace AppBundle\Form;

use AppBundle\Entity\User;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\Form\Extension\Core\Type\RepeatedType;
use Symfony\Component\Form\Extension\Core\Type\PasswordType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class RegistrationType extends AbstractType
{
	public function buildForm(
		FormBuilderInterface $builder,
		array $options
	)
	{
		$builder
			->add('username', TextType::class)
			->add('plainPassword', RepeatedType::class, [
				'invalid_message' => 'Fields doesn\'t match.'
				'type' =>		PasswordType::class
			])
		;
	}

	public function configureOptions(OptionsResolver $resolver)
	{
		$resolver->setDefault('data_class', User::class);
	}
}
```

Il est intéressant d'ajouter des annotations de validation telles que `@Symfony\Component\Validator\Constraints\NotBlank()` sur les propriétés `username` et `plainPassword`, qui permettra de s'assurer qu'elles ne soient pas vides lors de la soumission du formulaire.

Créons ensuite un template Twig `security/registration.html.twig`, ayant comme variable une instance de `FormView` obtenue à partir de `RegistrationType`. Il s'agit d'afficher le formulaire d'inscription. Nous pouvons maintenant implémenter l'action "register" associé à l'URL `/register`, chargée de rendre et de traiter le formulaire d'inscription, ainsi que de persister l'instance de `User` à la soumission du formulaire valide (c'est-à-dire enregistrer l'inscription) :

```
namespace AppBundle\Controller;

use AppBundle\Entity\User;
use AppBundle\Form\RegistrationType;
use Doctrine\ORM\EntityManagerInterface;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class SecurityController extends Controller
{
	/**
	* @Route("/register", name="security_register")
	*/
	public function registerAction(
		Request $request,
		EntityManagerInterface $manager
	)
	{
		$user = new User();
		$form = $this->createForm(RegistrationType::class, $user)
			->handleRequest($request);
		if ($form->isSubmitted() && $form->isValid()) {
			$manager->persist($user);
			$manager->flush($user);
			$this->addFlash(
				'success',
				'You\'ve been register, and can now log in'
			);

			return $this->render('app_home.html.twig');
		}

		return $this->render('security/registration.html.twig', [
			'form' => $form->createView()
		]);
	}
}
```

Il reste un dernier point à régler avant d'obtenir un mécanisme d'inscription complet : l'encodage du mot de passe utilisateur avant la persistance en base de donnée. Il suffit pour cela de créer un *listener* ou un *subscriber* actif sur l'événement Doctrine `prePersist`, auquel on injecte une dépendance avec l'encodeur de Symfony `Symfony\Component\Security\Core\Encoder\UserPasswordEncoder` :

```
namespace AppBundle\Event;

use AppBundle\Entity\User;
use Doctrine\ORM\Event\LifecycleEventArgs;
use Symfony\Component\Security\Core\Encoder\UserPasswordEncoder;

class HashUserPasswordListener
{
	private $encoder;

	public function __construct(UserPasswordEncoder $encoder)
	{
		$this->encoder = $encoder;
	}

	public function prePersist(LifecycleEventArgs $args)
	{
		$user = $args->getEntity();
		if ($user instanceof User && $user->getPlainPassword()) {
			$cypher = $this->encoder->encodePassword(
				$user,
				$user->getPlainPassword()
			);
			$user->setPassword($cypher);
		}
	}
}
```

Il ne faut pas oublier d'enregistrer ce service créé dans `config/services.yaml` :

```
services:
	# ...
	App\Event\HashUserPasswordListener:
		tags:
			- { name: doctrine.event_listener, event: prePersist }
```

Notre système d'inscription est désormais pleinement opérationnel.

Connexion par formulaire traditionnel

Nous allons dans cette section nous intéresser à l'implémentation d'un formulaire de connexion traditionnel, auquel nous devrons renseigner un nom d'utilisateur et un mot de passe pour nous authentifier. Configurons d'abord `security.yaml` pour indiquer que nous voulons nous connecter via un formulaire :

```
security:
      # ...
      firewalls:
            main:
                  anonymous: ~
                  form_login:
                        login_path: security_login
                        check_path: security_login
```

Dans la définition de notre pare-feu, nous renseigner la clé `form_login` avec 2 paramètres :

- `login_path` : la route associée au formulaire de connexion
- `login_check` : la route vers laquelle redirigera le formulaire de connexion, c'est à cette route que la demande authentification sera validée ou refusée.

Ici, nous indiquons la même route pour ces 2 clés.
Ajoutons maintenant l'action "login" chargée de rendre le formulaire de connexion (il faut impérativement s'assurer que cette route est accessible anonymement) :

```
namespace AppBundle\Controller;
# ...
use Symfony\Component\Security\Http\Authentication\AuthenticationUtils;

class SecurityController extends Controller
{
      /**
      * @Route("/login", name="security_login")
      */
      public function loginAction(
            Request $request,
            AuthenticationUtils $utils
      )
      {
            $error = $utils->getLastAuthenticationError();
            $lastUsername = $utils->getLastUsername();

            return $this->render('security/login.html.twig', [
                  'last_username' => $lastUsername,
                  'error'             => $error
            ]);
      }
}
```

Symfony se chargera automatiquement de traiter la soumission du formulaire. Si la soumission génère des erreurs (nom d'utilisateur ou mot de passe incorrect), le contrôleur récupère via le service `AuthenticationUtils` l'erreur ainsi que le nom d'utilisateur renseigné à la soumission, de sorte à les inclure dans le template Twig. La seule fonction de l'action "login" est de rendre le formulaire de connexion.
Créons le template Twig incluant le formulaire. Les *inputs* représentant le nom d'utilisateur et le mot de passe doivent être nommées respectivement `_username` et `_password` :

```
# ...
{% if error %}
      <div>{{ error.messageKey|trans(error.messageData, 'security') }}</div>
{% endif %}

<form action="{{ path('login') }}" 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" />
<button type="submit">login</button>
</form>
```

La route correspondant à l'attribut action du *tag* `<form>` n'est autre que celle renseignée pour la clé `form_check`. C'est à cette route que Symfony vérifiera automatiquement la validité des identifiants soumis. La connexion est désormais opérationnelle.
La clé de configuration de `security.yaml` `logout` est intéressante. Elle se positionne directement sous le nom de notre pare-feu, et permet d'activer la fonction de déconnexion :

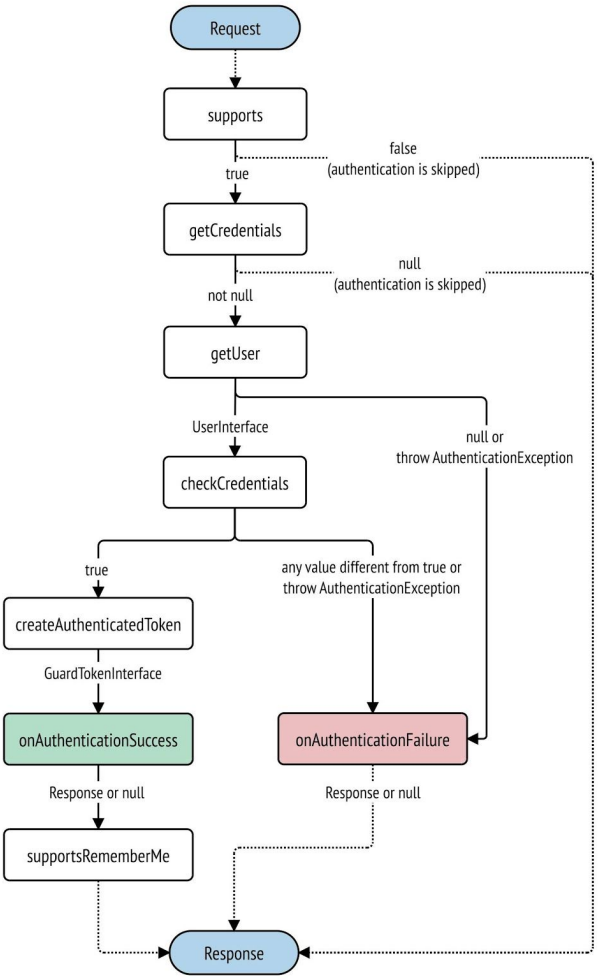
```
security:
      # ...
      firewalls:
      main:
            # ...
            logout:
                  path: /logout
      # ...
```

Il faut maintenant, pour pouvoir se déconnecter de l'application, ajouter une route associée à l'URL `/logout` et une action "logout" vide : le contenu de cette méthode ne sera jamais exécuté :

```
# ...
/**
* @Route("/logout", name="security_logout")
*/
public function logoutAction()
{
      throw new Exception();
}
```

Maîtriser le processus de connexion

Le formulaire de connexion traditionnel, présenté dans la section précédente, est relativement simple à mettre en place. Cependant Symfony automatise le traitement, ce qui masque le fonctionnement interne de la connexion. Nous allons ici voir plus en détail les différentes étapes suivant la soumission d'un formulaire de connexion, afin obtenir une vision d'ensemble du processus d'authentification. Pour ce faire, nous allons customiser le composant chargé de traiter la demande de connexion : le composant Guard.
Voici, dans les grandes lignes, la philosophie sur laquelle s'appuie le composant Guard : à chaque requête d'une URL placée sous un pare-feu défini dans `security.yaml`, le composant Sécurité va déterminer, en appelant la méthode `supports` du Guard, s'il s'agit d'une soumission de formulaire de connexion. Si non, l'authentification est ignorée, et Symfony traite normalement la requête. Si oui, Symfony va demander au Guard les *inputs* soumises, en appelant sa méthode `getCredentials` : `_username` et `_password`. Si ces elles sont renseignées, Symfony appelle la méthode `getUser` du Guard pour tenter de récupérer l'entité utilisateur. Si le nom d'utilisateur ne correspond à aucun utilisateur, Symfony va indiquer une erreur au service `AuthenticationUtils` (utilisé dans l'action "login" pour justement récupérer cette erreur ainsi que le nom d'utilisateur saisi). Si une instance UserInterface est retournée par `getUser`, Symfony appelle la méthode `checkCredentials` du Guard, qui va alors déterminer la validité du mot de passe. Si c'est correct, Symfony considère l'utilisateur comme authentifié, et place l'instance `UserInterface` sérialisée en session. Le client s'est alors connectée sur l'application. Voici un schéma résumant le mécanisme :



Créons notre propre Guard. Pour se faire, notre classe doit hériter de la classe abstraite `Symfony\Component\Security\Guard\AbstractGuardAuthenticator`. Nous allons ici avoir 2 injections de dépendances (le Guard doit être reconnu comme service) :

- `Symfony\Component\Form\FormFactory`, qui va nous servir à récupérer les données soumises dans le formulaire de connexion : `_username` et `_password`.
- `Symfony\Component\Security\Core\Encoder\UserPasswordEncoder`, qui va permettre d'encoder le mot de passe soumis `_password`, afin de vérifier son authenticité.

Il faut configurer le Guard sous la clé correspondant à notre pare-feu dans `security.yaml` (nous n'avons plus besoin de la clé `login_form`, qui permet le Guard par défaut de Symfony) :

```
security:
      # ..
      firewalls:
            main:
                  anonymous: ~
                  logout:
                        path: /logout
                  guard:
                        authenticators:
                              - AppBundle\Security\FormLoginAuthenticator
      # ...
```

Une implémentation possible est la suivante (le constructeur et les propriétés de la classe sont omises) :

```php
namespace AppBundle\Security;
# ...
class FormLoginAuthenticator extends AbstractFormLoginAuthenticator
{
    # ...
    public function supports(Request $request)
    {
        return $request->getPathInfo() == $this->router->generate('security_login')
            && $request->getMethod() == 'POST';
    }

    public function getCredentials(Request $request)
    {
        $data = $this->formFactory->createBuilder()
            ->add('_username', TextType::class)
            ->add('_password', TextType::class)
            ->getForm()
            ->handleRequest($request)
            ->getData();
        $request->getSession()->set(
            Security::LAST_USERNAME,
            $data['_username']
        );

        return $data;
    }

    public function getUser($credentials, UserProviderInterface $provider)
    {
        try {
            return $provider->loadUserByUsername($credentials['_username'];
        } catch (UsernameNotFoundException $e) {
            throw new CustomUserMessageAuthenticationException('Not Found');
        }
    }

    public function checkCredentials($credentials, UserInterface $user)
    {
        $isValid = $this->encoder->isPasswordValid(
            $user,
            $credentials['_password']
        );
        if (!$isValid) {
            throw new CustomUserMessageAuthenticationException('invalid');
        }

        return true;
    }
}
```

Conclusion

Le système de sécurité du *framework* Symfony est puissant et modulable, et sa configuration peut parfois être complexe. Cependant, nous avons pu, au travers de ce présent document, nous familiariser avec les mécanismes sur lesquels il s'appuie, et acquérir les connaissances nécessaire pour mettre en place un formulaire de connexion. Bien d'autres aspects du composant sécurité sont à découvrir, et la documentation officielle de Symfony est complète à ce sujet.