



Simon GAUTIER



Doctrine

Evènements et extensions

Les évènements Doctrine

Services qui écoutent les évènements Doctrine

Extensions Doctrine

Les évènements Doctrine

- Egalement appelés lifecycle callbacks
 - Associés au cycle de vie des entités qu'on manipule avec Doctrine
- Un évènement, c'est du fonctionnel, présent dans une méthode de l'entité qu'on exécute à des moments précis :
 - Avant ou après la sauvegarde d'une entité, avant ou après la suppression d'une entité, ...
- Quel intérêt ? Systématiser des actions
 - Exemple : après la suppression d'une entité, je veux toujours effectuer une action. Impossible de se dire qu'on y pensera à chaque fois qu'on fera une suppression. On utilise dans ce cas un évènement

Comment définir un évènement ?

- Etape 1 : « dire » à l'entité qu'elle contient des callbacks

- Une annotation de classe à ajouter :

```
/**
 * @ORM\Entity(repositoryClass="App\Repository\VehicleRepository")
 * @ORM\HasLifecycleCallbacks()
 */
class Vehicle
```

- Etape 2 : définir un callback (une méthode sans paramètres) :

- Quoi exécuter ?

```
public function forceDescription()
{
    if(is_null($this->description)) {
        // Uniquement si la description n'est pas renseignée, en définir une
        $this->description = "Le véhicule {$this->plate} a {$this->mileage} km";
    }
}
```

- Etape 3 : définir les évènements associés au callback

- Quand exécuter le callback ?
 - En l'occurrence, avant le persist de l'entité
 - Une annotation à ajouter sur le callback :

```
/**
 * @ORM\PrePersist()
 */
public function forceDescription()
```

Comment définir un évènement ?

- Illustration de l'exemple :

```
$vehicle = new Vehicle();  
$vehicle->setPlate('BB-111-BB')->setMileage(17500)->setPrice(14500)->setManufactureDate(new \DateTime('2015-06-25'));  
$em->persist($vehicle);  
$em->flush();
```

1	0.04 ms	"START TRANSACTION" Parameters: [] View formatted query View runnable query Explain query
2	36.97 ms	INSERT INTO vehicle (plate, mileage, price, description, manu_date, vehicle_security_id, model_id) VALUES (?, ?, ?, ?, ?, ?, ?) Parameters: [1 => "BB-111-BB" 2 => 17500 3 => 14500.0 4 => "Le véhicule BB-111-BB a 17500 km" 5 => "2015-06-25 00:00:00" 6 => null 7 => null] View formatted query View runnable query Explain query
3	204.68 ms	"COMMIT" Parameters: []

Liste des évènements disponibles

- **PrePersist :**

- Déclenché juste avant que l'EntityManager ne persiste effectivement l'entité (avant l'appel à la méthode **persist()** sur l'EntityManager) → les actions faites dans le callbacks seront persistées en base de données
- Ne se déclenche que pour les entités nouvellement créées → pas possible d'utiliser l'id de l'entité puisqu'il n'a pas encore été généré

- **PostPersist :**

- Déclenché juste après que l'EntityManager ait effectivement persisté l'entité.
Attention : ce n'est pas après l'appel à la méthode **persist()** mais après l'appel à la méthode **flush()** !
- Il est possible d'accéder à l'id de l'entité
- Les actions effectuées sur l'entité ne sont pas persistées en base de données

Liste des évènements disponibles

- **PreUpdate :**

- Déclenché juste avant que l'EntityManager ne modifie une entité → juste avant l'exécution du **flush()**
- Ne se déclenche que pour des objets qui sont déjà présents en base de données → il est possible d'accéder à l'id de l'entité
- Ne se déclenchera pas si l'objet n'a pas été modifié pour au moins un de ses attributs, même si on demande un **flush()**

- **PostUpdate :**

- Déclenché juste après que l'EntityManager ait effectivement modifié une entité → juste après l'exécution du **flush()**
- Il est possible d'accéder à l'id de l'entité
- Les actions effectuées sur l'entité ne sont pas persistées en base de données

Liste des évènements disponibles

- **PreRemove :**

- Déclenché juste avant que l'EntityManager ne supprime une entité → juste avant l'exécution du **flush()** qui suit le **remove()** demandé
- L'entité n'est pas encore supprimée → si par exemple on supprime des fichiers associés à cette entité et qu'ensuite le **flush()** échoue dans la suppression de l'entité, on aura perdu ces fichiers

- **PostRemove :**

- Déclenché juste après la suppression effective d'une entité
- L'id n'est plus accessible

Liste des évènements disponibles

- **PostLoad :**

- Déclenché juste après le chargement de l'entité par l'EntityManager
- Après un **find()** sur le Repository ou un **refresh()** sur l'EntityManager

- **ATTENTION**

- Aucun de ces évènements n'est déclenché si on fait du DQL
- Aucun de ces évènements n'est déclenché en passant par le QueryBuilder

Services qui écoutent les événements Doctrine

- Inconvénient des événements Doctrine présentés : ils n'accèdent pas à d'autres informations qu'à l'entité elle-même
- Exemple : vouloir envoyer un mail à la création d'une entité ➔ on a besoin du service **mailer**
- Solution : il est possible d'exécuter des services Symfony pour chaque événement du cycle de vie des entités (les mêmes événements que ceux évoqués précédemment)
 - Attention : Doctrine exécutera le service pour toutes les entités ➔ il est donc nécessaire de tester dans le service si l'entité traitée est du bon type (sauf si l'action doit être effectuée pour toutes les entités)

Services qui écoutent les événements

Doctrine

- Exemple de problématique :
 - On souhaite envoyer un mail dès qu'une entité Vehicle est créée sans être rattachée à un VehicleModel
- Conceptuellement, commencer par séparer les fonctionnalités :
 - D'une part, l'envoi du mail d'alerte
 - D'autre part, le fait de détecter qu'un Vehicle est créé sans être rattaché à un VehicleModel
- On va créer 2 services :
 - Un service qui envoie le mail pour un Vehicle
 - Un service qui détecte que le Vehicle a été créé sans être rattaché à un VehicleModel
- Pourquoi séparer les services ? Par exemple car on peut imaginer réutiliser dans un autre contexte le service qui envoie un mail d'alerte pour un Vehicle
 - ➔ Nous aurions pu faire un seul service mais il est bien de comprendre la logique de séparation

Service d'envoi du mail pour un véhicule

```
<?php
namespace App\Vehicle;

use App\Entity\Vehicle;

class Mailer
{
    /** @var \Swift_Mailer */
    private $mailer;

    /**
     * Mailer constructor.
     * @param $mailer
     */
    public function __construct(\Swift_Mailer $mailer)
    {
        $this->mailer = $mailer;
    }

    public function sendMissingModel(Vehicle $vehicle)
    {
        $message = new \Swift_Message(
            "Vehicule {$vehicle->getPlate()} sans modèle",
            "Attention : le véhicule {$vehicle->getPlate()} a été créé sans modèle. Merci de faire les corrections nécessaires"
        );

        $message
            ->addTo('simon.gautier@emilfrey.fr') // TODO : passer par une variable !
            ->addFrom('simon.gautier@emilfrey.fr'); // TODO : passer par une variable !

        $this->mailer->send($message);
    }
}
```

Service qui détecte l'évènement qui nous intéresse

```
<?php
namespace App\Vehicle;

use App\Entity\Vehicle;
use App\Entity\VehicleModel;
use Doctrine\ORM\Event\LifecycleEventArgs;

class Creation
{
    /** @var Mailer */
    private $vehicleMailer;

    /**
     * Creation constructor.
     * @param Mailer $vehicleMailer
     */
    public function __construct(Mailer $vehicleMailer)
    {
        $this->vehicleMailer = $vehicleMailer;
    }

    public function postPersist(LifecycleEventArgs $args)
    {
        // La méthode doit porter le nom de l'évènement déclaré dans le services.yaml
        // Noter que le paramètre LifecycleEventArgs permet également d'accéder à l'EntityManager

        $entity = $args->getEntity();

        if(!$entity instanceof Vehicle) {
            // Ne rien faire s'il ne s'agit pas d'une entité Vehicle ==> ne pas oublier ce test !
            return;
        }

        if(is_object($entity->getModel()) && $entity->getModel() instanceof VehicleModel) {
            // Ne rien faire dans ce cas car le modèle est bien renseigné
            return;
        }

        // Faire appel au service Mailer qui enverra l'alerte
        $this->vehicleMailer->sendMissingModel($entity);
    }
}
```

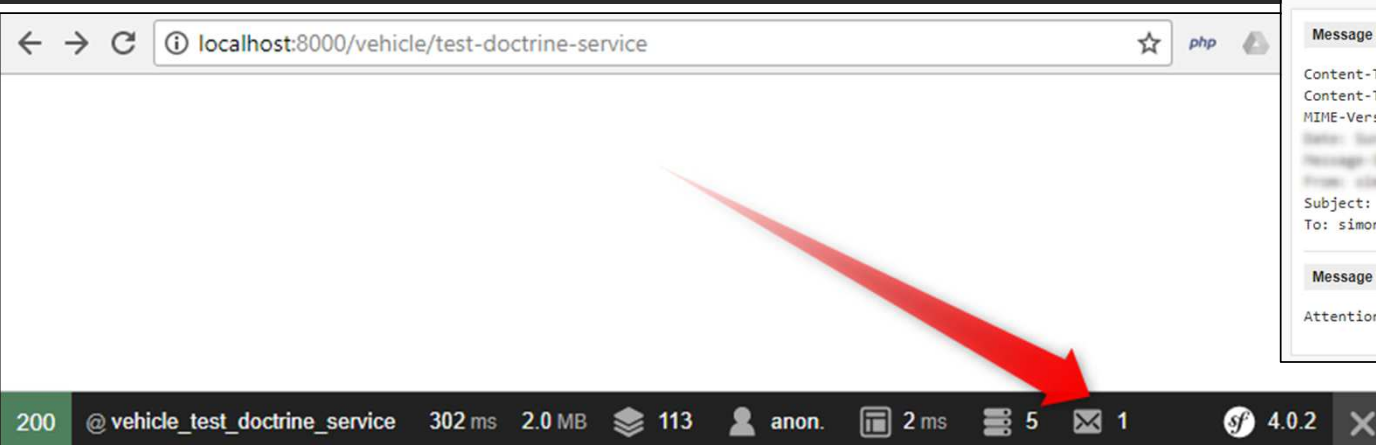
```
services:
    # ...
    App\Vehicle\Creation:
        # Utiliser le tag doctrine.event_listener ainsi
        # que l'event Doctrine concerné (postPersist ici)
        tags:
            - { name: doctrine.event_listener, event: postPersist }
```

Testons nos 2 services !

```
public function testDoctrineService(EntityManagerInterface $em)
{
    $vehicle1 = new Vehicle(); // Un véhicule sans modèle
    $vehicle1->setPlate('AA-123-BB')->setMileage(17500)->setPrice(14500)->setManufactureDate(new \DateTime('2015-06-25'));

    $vehicle2 = new Vehicle(); // Un véhicule avec modèle
    $vehicle2->setPlate('BB-321-AA')->setMileage(17500)->setPrice(14500)->setManufactureDate(new \DateTime('2015-06-25'));
    $model = new VehicleModel();
    $model->setName('Clio')->setMake('Renault');
    $vehicle2->setModel($model);

    $em->persist($vehicle1);
    $em->persist($vehicle2);
    $em->persist($model);
    $em->flush();
    return new Response('<body></body>');
}
```



← → ↻ ⓘ localhost:8000/vehicle/test-doctrine-service ☆ php

200 @ vehicle_test_doctrine_service 302 ms 2.0 MB 113 anon. 2 ms 5 1 4.0.2

E-mails

1

spooled message

E-mail details

Message headers

Content-Transfer-Encoding: quoted-printable
Content-Type: text/plain; charset=utf-8
MIME-Version: 1.0
Date: Sun, 24 Jan 2016 17:38:58 +0000
Message-ID: <1453474811.17717401000000000000@localhost>
From: simon.gautier@pgamotors.com
Subject: Vehicule AA-123-BB sans =?utf-8?Q?mod=C3=A8le?=
To: simon.gautier@pgamotors.com

Message body

Attention : le véhicule AA-123-BB a été créé sans modèle. Merci de faire les corrections nécessaires

Listener VS subscriber

- L'exemple présente un doctrine.event_listener. Il existe une alternative proche : le doctrine.event_subscriber, cf. doc : symfony.com/doc/current/doctrine/event_listeners_subscribers.html
- Principale différence : c'est le subscriber lui-même qui précise les évènements Doctrine qu'il écoute :

```
services:
    # ...
    App\Vehicle\Creation:
        tags:
            - { name: doctrine.event_subscriber }
```

Déclaration d'un subscriber

```
use Doctrine\Common\EventSubscriber;

class Creation implements EventSubscriber
{
    public function getSubscribedEvents()
    {
        return array(
            'postPersist',
        );
    }
    // ...
}
```

Le service doit implémenter la classe EventSubscriber et donc définir la méthode getSubscribedEvents(). Le reste du code est inchangé.

Les extensions Doctrine

- Doctrine propose plusieurs extensions pour des fonctionnalités assez récurrentes
 - Objectif : nous éviter d'avoir à réinventer la roue
- Pour profiter de ces extensions Doctrine, il faut installer le bundle **StofDoctrineExtensionsBundle** :
<https://packagist.org/packages/stof/doctrine-extensions-bundle>
- L'installation du bundle est classique :
composer req stof/doctrine-extensions-bundle
- Il faut ensuite prendre en compte la procédure d'installation qui décrit certaines subtilités en fonction des extensions dont on aura besoin :
<https://github.com/stof/StofDoctrineExtensionsBundle/blob/master/Resources/doc/index.rst>

Les extensions Doctrine disponibles

- Tree : facilite la gestion des arbres
 - Ajoute des méthodes spécifiques dans le repository
- Translatable : ajoute une logique de traduction pour les attributs de nos entités
 - Rend transparente la gestion des traductions pour les attributs
 - Charge automatiquement les traductions de la locale courante
- Sluggable : génère automatiquement un slug à partir d'attributs spécifiés
 - Exemple de slug : « Générer un slug » ➔ « generer-un-slug »
 - Utile par exemple pour faire de la réécriture d'URL

Les extensions Doctrine disponibles

- Timestampable : automatise la mise à jour d'attributs de type date dans nos entités (ce qu'on a fait manuellement lors de précédents TPs 😊)
 - Lors de la mise à jour / la création de l'entité
 - Lors de la mise à jour d'un attribut en particulier
- Blameable : permet d'associer l'utilisateur courant à une entité afin de savoir « qui fait quoi »
 - Savoir qui a créé l'entité, qui a modifié l'entité, qui a modifié tel attribut de l'entité, ...
 - Très similaire à Timestampable mais pour des utilisateurs et pas des dates
 - Si l'attribut « utilisateur qui a... » est une string ➔ le username sera stocké
 - Si l'attribut « utilisateur qui a... » est une association avec une entité user ➔ le lien avec cette entité sera fait

Les extensions Doctrine disponibles

- Loggable :
 - Permet de conserver les différentes versions de nos entités
 - Offre des outils de gestion des versions
- Sortable : permet de gérer des attributs de positions au sein des entités et de les maintenir à jour
- Softdeleteable : permet de supprimer logiquement (et pas physiquement) une entité
 - Gère un flag de suppression
 - Propose des filtres automatiques lors des SELECT pour ne pas remonter les éléments supprimés

Les extensions Doctrine disponibles

- Uploadable :
 - Offre des outils pour gérer l'enregistrement de fichiers associés avec des entités
 - Inclut la gestion automatique des déplacements et des suppressions des fichiers
- IpTraceable : assigner l'IP de l'utilisateur courant à un attribut
 - Même principe que Timestampable ou Blameable mais pour le stockage de l'IP de l'utilisateur qui a créé / modifié l'entité ou un attribut spécifique de l'entité

Exemple de l'utilisation de l'extension Sluggable

Définition d'un slug pour le modèle d'un véhicule

- Etape 1 : activer l'extension dans `config/packages/stof_doctrine_extensions.yaml` :

```
stof_doctrine_extensions:
    # ...
    orm:
        default:
            sluggable: true
```

- Etape 2 : ajouter l'attribut de classe qui sera le slug :

```
/**
 * @var string
 * @ORM\Column(unique=true)
 * @Gedmo\Slug(fields={"make", "name"})
 */
private $slug;
```

- ATTENTION : nécessite : « `use Gedmo\Mapping\Annotation as Gedmo;` »
- On choisit ici de générer le slug à partir des 2 attributs make et name
- L'attribut est un attribut « comme un autre », on le choisit juste « unique »
- Puis, comme d'habitude :
 - Ajouter le getter / le setter
 - Mettre à jour la base de données. Remarque : plante lors de la mise à jour de la base de données car le « unique » refuse d'avoir plusieurs attributs avec la valeur vide (→ requiert pour le test de vider la table des modèles de véhicules)


Exemple de l'utilisation de l'extension Sluggable

Définition d'un slug pour le modèle d'un véhicule

- Test de l'ajout d'un modèle :

#▲	Time	Info
1	1.18 ms	<pre>SELECT v0_.slug AS slug_0 FROM vehicle_model v0_ WHERE v0_.slug LIKE ?</pre> <p>Parameters:</p> <pre>["renault-megane%"]</pre> <p>View formatted query View runnable query Explain query</p>
2	0.02 ms	<pre>"START TRANSACTION"</pre> <p>Parameters:</p> <pre>[]</pre> <p>View formatted query View runnable query Explain query</p>
3	13.89 ms	<pre>INSERT INTO vehicle_model (name, make, slug) VALUES (?, ?, ?)</pre> <p>Parameters:</p> <pre>[1 => "Mégane" 2 => "Renault" 3 => "renault-megane"]</pre> <p>View formatted query View runnable query Explain query</p>
4	33.78 ms	<pre>"COMMIT"</pre> <p>Parameters:</p> <pre>[]</pre> <p>View formatted query View runnable query Explain query</p>

```
public function addModelAction(EntityManagerInterface $em)
{
    $model = new VehicleModel();
    $model->setName('Mégane')->setMake('Renault');
    $em->persist($model);
    $em->flush();
    return new Response('<body></body>');
}
```



TP Evènements et extensions Doctrine



- Evènement Doctrine simple
 - Créer un évènement Doctrine simple qui renseigne le champ « **author** » d'un article avec la valeur « anonymous » si celui-ci n'est pas renseigné lors de la création d'une entité Article
 - Tester avec le formulaire de création d'un article
- Evènements déclencheurs de services
 - Créer un service qui envoie un mail pour une entité Article (dépendance au service **mailer**)
 - Créer un service qui lors de la mise à jour d'un article envoie un mail dès que l'article a été consulté 100, 200, 300, ... fois (dépendance au service qu'on vient de créer) → vous pouvez faire 10, 20, 30 pour éviter d'actualiser des centaines de fois la consultation d'un article 😊
 - Dans l'action **viewAction()** du contrôleur des articles, penser à incrémenter le nombre de consultations d'un article à chaque consultation
- Extensions Doctrine Sluggable
 - Implémenter l'extension Sluggable : créer un attribut « slug » unique sur l'entité Article et l'alimenter à partir de l'attribut « **title** ». Tester avec le formulaire de création d'un article
 - Créer une action **viewSlugAction()** avec sa route qui attend en paramètre non pas un ID mais un slug



Simon GAUTIER



Sécurité – Les voters

Les voters – Gestion avancée des autorisations

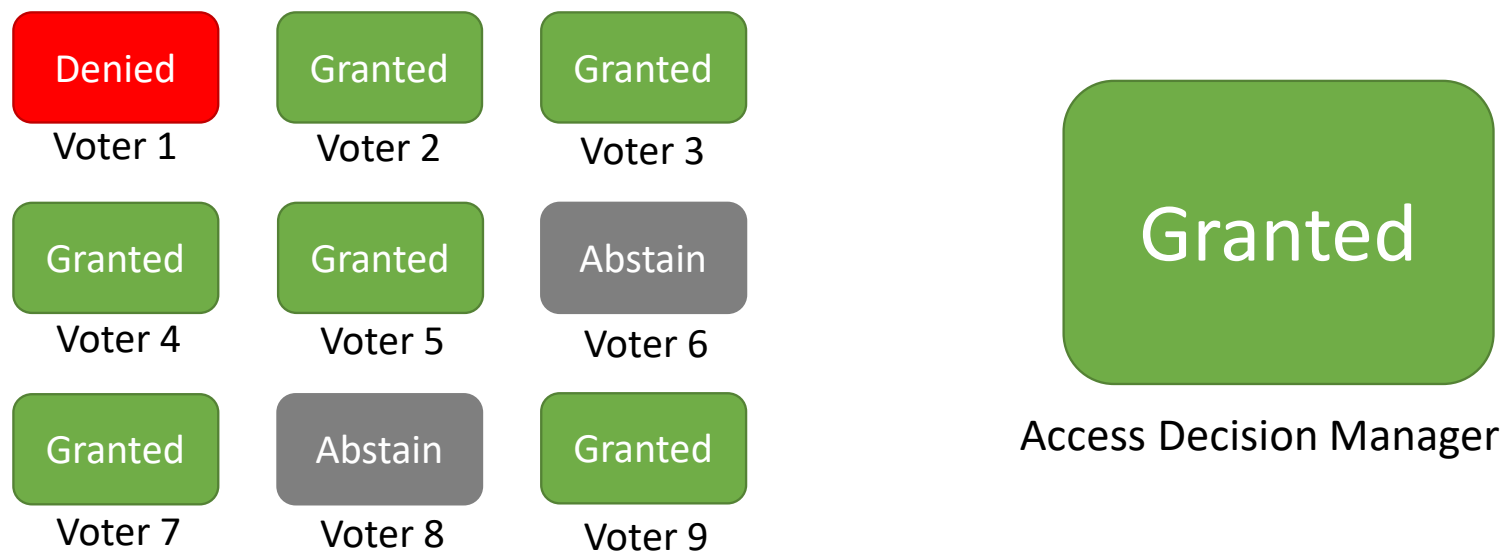
- Focus sur la méthode **isGranted()** :
 - **\$token** : les informations de connexion de l'utilisateur
 - **\$attributes** : ce sur quoi porte le test d'authentification → est-ce que l'utilisateur a le droit de faire telle(s) action(s) ?
 - **\$subject** : un objet dont on peut avoir besoin pour contrôler les droits. Ex : un article dans le cas d'un blog pour savoir si l'utilisateur a le droit de modifier l'article
 - **decide()** : prend la décision afin de savoir si les droits sont accordés ou non → s'appuie sur l'Access Decision Manager, lequel s'appuie sur les voters

```
final public function isGranted($attributes, $subject = null)
{
    if (null === ($token = $this->tokenStorage->getToken())) {
        throw new AuthenticationCredentialsNotFoundException('The token storage contains no authentication token.
        One possible reason may be that there is no firewall configured for this URL.');
```


Les voters – Gestion avancée des autorisations

Quelques définitions

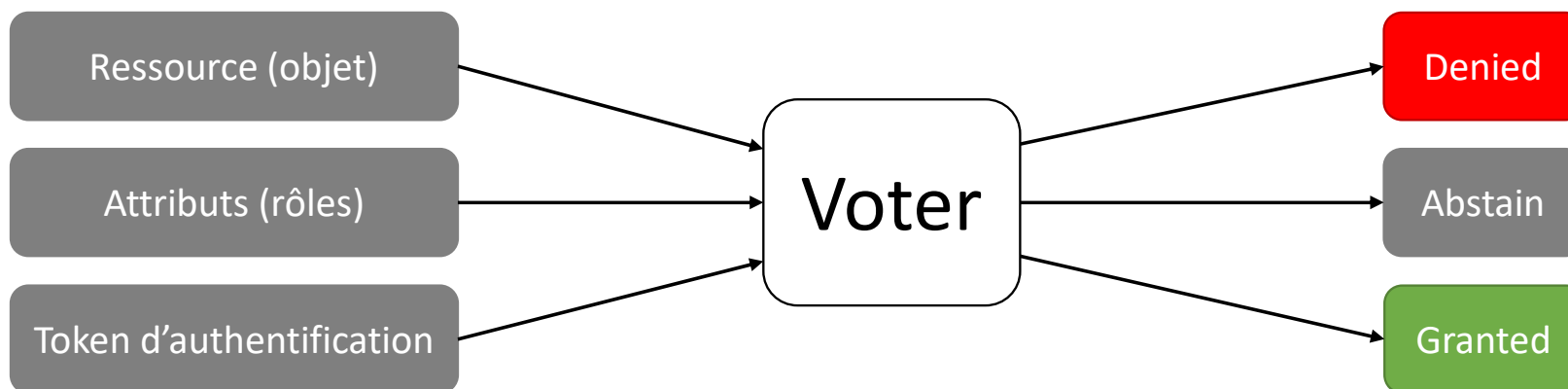
- Access Decision Manager : service chargé d'accorder ou refuser l'accès à un utilisateur final en fonction d'une stratégie donnée
- Stratégie : décide si l'utilisateur est autorisé à accéder à une ressource ou à effectuer une action. La décision est confiée à une liste de **voters**



Les voters – Gestion avancée des autorisations

Qu'est-ce qu'un voter ?

- Un voter est un objet qui décide si oui ou non, un utilisateur représenté par un token d'authentification peut accéder à une ressource protégée dans un contexte donné
- Un voter peut accorder l'accès, le refuser ou s'abstenir s'il n'est pas en mesure de prendre une décision



Les voters – Gestion avancée des autorisations

Stratégies disponibles

- Unanime : Accorde l'accès si tous les voters accordent l'accès
- Consensus : Accorde l'accès s'il y a plus de voters qui accordent l'accès que de voters qui le refusent
- Affirmatif (fonctionnement standard de Symfony) : Accorde l'accès si au moins un voter accorde l'accès
- La stratégie est définie dans le **security.yaml** :

```
security:  
  access_decision_manager:  
    strategy: unanimous
```

```
security:  
  access_decision_manager:  
    strategy: consensus
```

```
security:  
  access_decision_manager:  
    strategy: affirmative
```

Les voters – Gestion avancée des autorisations

Zoom sur la VoterInterface que tous les voters implémentent

```
/**
 * VoterInterface is the interface implemented by all voters.
 */
interface VoterInterface
{
    const ACCESS_GRANTED = 1;
    const ACCESS_ABSTAIN = 0;
    const ACCESS_DENIED = -1;

    /**
     * Returns the vote for the given parameters.
     *
     * This method must return one of the following constants:
     * ACCESS_GRANTED, ACCESS_DENIED, or ACCESS_ABSTAIN.
     *
     * @param TokenInterface $token      A TokenInterface instance
     * @param mixed          $subject    The subject to secure
     * @param array          $attributes An array of attributes associated with the method being invoked
     *
     * @return int either ACCESS_GRANTED, ACCESS_ABSTAIN, or ACCESS_DENIED
     */
    public function vote(TokenInterface $token, $subject, array $attributes);
}
```

On retrouve les 3 notions présentées précédemment :

- **\$token**
- **\$subject**
- **\$attributes :**

Les voters

- Zoom sur le voter RoleVoter :
 - Le RoleVoter vote si au moins un **\$attribute** commence avec le préfixe « **ROLE_** ». Sinon, il s'abstient
 - Noter que la variable **\$subject** n'est pas utilisée dans ce voter
 - **\$token** est lui utilisé pour extraire les rôles auxquels l'utilisateur a droit afin de le comparer au rôle testé

```
class RoleVoter implements VoterInterface
{
    private $prefix;

    public function __construct(string $prefix = 'ROLE_')
    {
        $this->prefix = $prefix;
    }

    /**
     * {@inheritdoc}
     */
    public function vote(TokenInterface $token, $subject, array $attributes)
    {
        $result = VoterInterface::ACCESS_ABSTAIN;
        $roles = $this->extractRoles($token);

        foreach ($attributes as $attribute) {
            if ($attribute instanceof Role) {
                $attribute = $attribute->getRole();
            }

            if (!is_string($attribute) || 0 !== strpos($attribute, $this->prefix)) {
                continue;
            }

            $result = VoterInterface::ACCESS_DENIED;
            foreach ($roles as $role) {
                if ($attribute === $role->getRole()) {
                    return VoterInterface::ACCESS_GRANTED;
                }
            }
        }

        return $result;
    }

    protected function extractRoles(TokenInterface $token)
    {
        return $token->getRoles();
    }
}
```

Classe permettant de faciliter
la création de voters :

1/2

La méthode **vote()** est déjà
définie :

- Si aucun **\$attribute** / **\$subject** ne matche, le voter s'abstient
- Dès qu'un triplet **\$attribute** / **\$subject** / **\$token** est OK, le voter donne l'accès
- Si au moins un **\$attribute** / **\$subject** matche mais qu'aucun triplet n'est OK, le voter refuse l'accès

```
/**
 * Voter is an abstract default implementation of a voter.
 *
 * @author Roman Marintšenko <inoryy@gmail.com>
 * @author Grégoire Pineau <lyrixx@lyrixx.info>
 */
abstract class Voter implements VoterInterface
{
    /**
     * {@inheritdoc}
     */
    public function vote(TokenInterface $token, $subject, array $attributes)
    {
        // abstain vote by default in case none of the attributes are supported
        $vote = self::ACCESS_ABSTAIN;

        foreach ($attributes as $attribute) {
            if (!$this->supports($attribute, $subject)) {
                continue;
            }

            // as soon as at least one attribute is supported, default is to deny access
            $vote = self::ACCESS_DENIED;

            if ($this->voteOnAttribute($attribute, $subject, $token)) {
                // grant access as soon as at least one attribute returns a positive response
                return self::ACCESS_GRANTED;
            }
        }

        return $vote;
    }
    // La suite sur la slide suivante ☺
}
```

Classe permettant de faciliter la création de voters : 2/2

```
/**
 * Determines if the attribute and subject are supported by this voter.
 *
 * @param string $attribute An attribute
 * @param mixed $subject The subject to secure, e.g. an object the user wants to access or any other PHP type
 *
 * @return bool True if the attribute and subject are supported, false otherwise
 */
abstract protected function supports($attribute, $subject);

/**
 * Perform a single access check operation on a given attribute, subject and token.
 * It is safe to assume that $attribute and $subject already passed the "supports()" method check.
 *
 * @param string $attribute
 * @param mixed $subject
 * @param TokenInterface $token
 *
 * @return bool
 */
abstract protected function voteOnAttribute($attribute, $subject, TokenInterface $token);&
```

Principe de mise en œuvre d'un voter :

- Créer une classe qui hérite de cette classe abstraite
- Implémenter les 2 méthodes **supports()** et **voteOnAttribute()**
 - **supports()** permet de savoir si le voter est concerné par le vote sur l'attribut et le sujet donnés
 - **voteOnAttribute()** permet de savoir si l'utilisateur a les droits pour un sujet et un attribut donnés

Les voters – Gestion avancée des autorisations

Mise en œuvre d'un voter basé sur la classe abstraite présentée

- Exemple concret sur un véhicule :
 - Seuls les véhicules avec une plaque d'immatriculation peuvent être consultés par tout le monde
 - Les véhicules peuvent être édités uniquement par leurs propriétaires respectifs
- Etapes préliminaires :
 - Créer une entité User → doit implémenter à minima l'interface **\Symfony\Component\Security\Core\User\UserInterface**
 - Paramétrer le **security.yaml** pour utiliser cette entité → ajouter un nouveau provider
 - Lier les véhicules à cette entité : Vehicle Many-To-One User
 - Lors de l'ajout d'un véhicule, sauvegarder l'utilisateur courant → via un service qui écoute un événement Doctrine


```

/**
 * @ORM\Entity(repositoryClass="App\Repository\UserRepository")
 */
class User implements UserInterface
{
    /**
     * @ORM\Id
     * @ORM\GeneratedValue
     * @ORM\Column(type="integer")
     */
    private $id;

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

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

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

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

    public function getSalt()
    {
        return null;
    }
}

```

```

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

public function eraseCredentials()
{
}

/**
 * @return mixed
 */
public function getId()
{
    return $this->id;
}

/**
 * @param string $username
 * @return User
 */
public function setUsername(string $username): User
{
    $this->username = $username;
    return $this;
}

/**
 * @param string $password
 * @return User
 */
public function setPassword(string $password): User
{
    $this->password = $password;
    return $this;
}
}

```

```

security:
    # ...
    encoders:
        App\Entity\User: bcrypt
        # ...

    providers:
        my_provider_entity:
            entity:
                class: App\Entity\User
                property: username
            # ...

    firewalls:
        # ...
        main:
            # ...
            provider: my_provider_entity

```

Etape préliminaire 1 :

Création d'une entité pour les utilisateurs + mise à jour du **security.yaml**

Les voters – Gestion avancée des autorisations

Mise en œuvre d'un voter basé sur la classe abstraite présentée

```
class Vehicle
{
    // ...
    /**
     * @var User
     * @ORM\ManyToOne(targetEntity="App\Entity\User")
     */
    private $user;

    // ...

    /**
     * @return mixed
     */
    public function getUser()
    {
        return $this->user;
    }

    /**
     * @param mixed $user
     * @return Vehicle
     */
    public function setUser($user)
    {
        $this->user = $user;
        return $this;
    }
    // ...
}
```

- Etape préliminaire 2 :
 - Lier les véhicules à l'entité User :
Vehicle Many-To-One User

Les voters – Gestion avancée des autorisations

Mise en œuvre d'un voter basé sur la classe abstraite présentée

```
class SaveUser implements EventSubscriber
{
    private $token;

    public function __construct(TokenStorageInterface $token)
    {
        $this->token = $token;
    }


    public function getSubscribedEvents()
    {
        return array(
            'prePersist',
        );
    }

    public function prePersist(LifecycleEventArgs $args)
    {
        $entity = $args->getEntity();

        if (!$entity instanceof Vehicle) {
            return;
        }


        if (null === $this->token
            || !is_object($token = $this->token->getToken())
            || !is_object($user = $token->getUser())) {
            return;
        }

        $entity->setUser($user);
    }
}
```



- Etape préliminaire 3 :

- Lors de l'ajout d'un véhicule, sauvegarder l'utilisateur courant → via un service qui écoute un évènement Doctrine
- Penser à déclarer l'évènement subscriber dans le **services.yaml** :



```
services:
    # ...
    App\Vehicle\SaveUser:
        tags:
            - { name: doctrine.event_subscriber }
```

Les voters – Gestion avancée des autorisations

Mise en œuvre d'un voter basé sur la classe abstraite présentée

```
<?php
namespace App\Security\Voter;

use App\Entity\User;
use App\Entity\Vehicle;
use Symfony\Component\Security\Core\Authentication\Token\TokenInterface;
use Symfony\Component\Security\Core\Authorization\Voter\Voter;

class VehicleVoter extends Voter
{
    protected function supports($attribute, $subject)
    {
        return $subject instanceof Vehicle && in_array($attribute, ['view', 'edit']);
    }

    protected function voteOnAttribute($attribute, $vehicle, TokenInterface $token)
    {
        // Les véhicules avec plaque d'immatriculation peuvent être consultés par tout le monde
        if ('view' === $attribute && $vehicle->getPlate()) {
            return true;
        }

        // Seul l'utilisateur qui a créé le véhicule peut le modifier
        $userId = $token->getUser()->getId();
        $owner = $vehicle->getUser();
        if ('edit' === $attribute && $owner instanceof User && $userId === $owner->getId()) {
            return true;
        }

        return false;
    }
}
```

Définition du voter

Prévoir également le paramétrage dans **services.yaml** sauf si l'autowire et l'autoconfigure sont à true :

```
services:
    # ...

    # La déclaration du service n'est
    # pas nécessaire si autowire: true
    # et autoconfigure: true
    App\Security\Voter\VehicleVoter:
        tags:
            - { name: security.voter }
```

Les voters – Gestion avancée des autorisations

Mise en œuvre d'un voter basé sur la classe abstraite présentée

```
/**
 * @Route("/vehicle/view/{id}", name="vehicle_view")
 * @param Vehicle $vehicle
 * @IsGranted("ROLE_USER")
 * @return Response
 */
public function viewAction(Vehicle $vehicle, Security $security)
{
    if (!$security->isGranted('view', $vehicle)) {
        throw new NotFoundException('Véhicule inconnu');
    }
    // ...
}

/**
 * @Route("/vehicle/edit/{id}", name="vehicle_edit")
 * @param Vehicle $vehicle
 * @IsGranted("ROLE_USER")
 * @return Response
 */
public function editAction(Vehicle $vehicle, Security $security)
{
    if (!$security->isGranted('edit', $vehicle)) {
        throw new AccessDeniedException("Vous n'avez pas le droit d'éditer ce véhicule");
    }
    // ...
}
```

Mise en application du voter
dans un contrôleur / dans
un template Twig

```
{% if is_granted('edit', vehicle) %}
    <a href="...">Editer le véhicule</a>
{% endif %}
```

TP Sécurité – Les voters



- Créer un voter permettant de vérifier les règles suivantes :
 - Seuls les articles publiés doivent pouvoir être consultés
 - Un article ne peut être édité que par celui qui l'a créé
- Etapes préliminaires identiques à celles présentées lors du cours :
 - Créer une entité User → doit implémenter à minima l'interface **\Symfony\Component\Security\Core\User\UserInterface**
 - Paramétrer le **security.yaml** pour utiliser cette entité → ajouter un nouveau provider
 - Lier les articles à cette entité : Article Many-To-One User
 - Lors de l'ajout d'un article, sauvegarder l'utilisateur courant → via un service qui écoute un évènement Doctrine