# - QueryBuilder-

Etend les possibilités de la classe Repository

C’est dans les classes Repository :

Exemple : Dans ThemeRepository, ajouter une méthode getAllThemes(pour l’exemple)

**public function** getAllThemes() {  
 // Recupérer l'instance de QueryBuilder. 't' est l'alias THEME car on est dans ThemeRepository  
 $qb = $this->createQueryBuilder('t');  
  
 $qb->select("t.name, count(p) as numerofPosts")  
 // 1ere argument est la variable qui stocke l'association  
 // le 2e argument représente l'alias de la 2e entité. En l'occurence c'est Post  
 ->innerJoin("t.posts", "p")  
 ->groupBy("t.id");  
  
 **return** $qb->getQuery();  
 }

A la fin de la méthode, faire un return de getQuery().

Puis dans l’action de notre Controller, appeler le getAllThemes via le repository. Voir code:

$repository = $this->getDoctrine()  
 ->getRepository("AppBundle:Theme");  
  
 $themeList = $repository->findAll();  
// Pour les tests du QueryBuilder  
$list = $repository->getAllThemes()->getArrayResult();

Voir doc à partir de la page 100

Installation de oro/doctrine-extensions

Utiliser certaines fonctions sql dans QueryBuilder car Doctrine ne le gère pas

composer require oro/doctrine-extensions

voir la doc :

<https://github.com/oroinc/doctrine-extensions>

pour la suite de l’installation pour la modification du fichier config.yml

Exemple 2 : Ajout d’une autre méthode dans le Repository avec QueryBuilder dans l’entité Post

**public function** getPostsGroupedByYear() {  
 $qb = $this->createQueryBuilder('p');  
  
 $qb->select('YEAR(p.createdAt) as yearPublished,  
 COUNT(p.id) as numberOfPosts ')  
 ->groupBy('YEAR(yearPublished)');  
  
 **return** $qb->getQuery()->getArrayResult();  
}

SLUG

stof/doctrine-extensions-bundle

Installation

composer require stof/doctrine-extensions-bundle

Un fois installé, il faut le référencer dans AppKernel.php

**new** \Stof\DoctrineExtensionsBundle\StofDoctrineExtensionsBundle(),

Dans l’Entité où on veut le SLUG, on fait un use :

**use** Gedmo\Mapping\Annotation **as** Gedmo;

En l’occurrence, dans l’entité POST

Ajouter une colonne dans l’entité post :

*/\*\*  
 \** ***@ORM\Column****(name="slug", type="string", length=255, unique=true)  
 \** ***@Gedmo\Slug****(fields={"author", "title"})  
 \** ***@var*** *string  
 \*/***private** $slug;

Fields={} : on définit la liste des champs qui vont constituer le slug

Dans config.yml, activer stof avec la clé suivante :

**stof\_doctrine\_extensions:  
 default\_locale:** fr\_FR  
 **orm:  
 default:  
 sluggable:** true

On fait : php bin/console doctrine:schema:drop –force

Puis : php bin/console doctrine:schema:create

Enfin : php bin/console doctrine:fixtures:load

Après il faut changer la route et les liens

# FORMULAIRE : Synfony

Création des formulaire : theme et Post  
Mise en forme bootstrap en modifiant la clé twig dans config.yml, en ajoutant la clé suivante :  
 form\_themes:  
 - "bootstrap\_3\_layout.html.twig"

Pour la création des forms, taper la commande suivante pour le post :

* php bin/console doctrine:generate:form AppBundle:Post

Cette commande nous crée, dans notre bundle, le dossier form avec le fichier PostType.php qui représente notre Entité post.

# CKEditor

composer require egeloen/ckeditor-bundle

Modifier le AppKernel

**new** Ivory\CKEditorBundle\IvoryCKEditorBundle(),

puis faire un assets: install

* Php bin console ckeditor:install
* Php bin console assets:install (sert à sopier dans le dossier web, tous les fichiers js et css exposés par le bundle

Ajouter les liens js dans les blocks {%block javascript %}

{% **block** javascripts %}  
 <script src="{{ asset('bower-components/jquery/dist/jquery.min.js') }}"></script>  
 <script src="{{ asset('bower-components/bootstrap/dist/js/bootstrap.min.js') }}"></script>

<script src="{{ asset('bundles/ivoryckeditor/ckeditor.js') }}"></script>

{% **endblock** %}

Dans notre PostType, remplacer TextareaType ::class par CKEditorType::*class*

Installer un bundle de purification : exercise/html purifier-bundle

* Composer require exercise/htmlpurifier-bundle

Ajouter dans AppKernel :

**new** Exercise\HTMLPurifierBundle\ExerciseHTMLPurifierBundle(),

dans le twig qui doit afficher le texte, faire un PIPE purifier

{{ post.text | purify }}

# Sécurité-Connexion

Pour protéger un espace de notre site

Commencer par modifier dans security.yml, la clé « providers » comme suit :

*# http://symfony.com/doc/current/security.html#b-configuring-how-users-are-loaded***providers:  
 in\_memory:  
 memory:  
 users:  
 admin:  
 password:** 123  
 **roles:** ROLE\_ADMIN

“Admin” est le login de notre user

« In\_memory » garde ces infos en mémoire

En rajoute aussi cette clé « encoders » pour le type d’encodage :

**encoders:  
 Symfony\Component\Security\Core\User\User:** plaintext

Dans “firewall”, on protège /admin/login, en permettant à tous d’arriver sur cette page:

*# Je protège la route /admin/login***admin\_login\_firewall:  
 pattern:** ^/admin/login$  
 **anonymous:** ~

Puis on protège les autres routes commencant par /admin

*# Je protège toutes les routes commencant par /admin***admin\_firewall:  
 pattern:** ^/admin  
 **provider:** in\_memory *# La clé fournisseur de donnée* **form\_login:  
 login\_path:** admin\_login *# Route pour afficher le formulaire de connexion* **check\_path:** admin\_login\_check *# Route pour exécution l'action sur le formulaire* **default\_target\_path:** admin\_home *# Route cible si action est ok*

## Modfication de routing.yml

**admin\_login\_check:** *# Nom de la route pour la vérification* **path:** /admin/login-check *# Chemin de la route pour la vérification*

## Dans le controller :

*/\*\*  
 \** ***@Route****("/", name="admin\_home")  
 \** ***@return*** *Response  
 \*/***public function** indexAction() {  
 **return** $this->render('admin/index.html.twig');  
}  
  
*/\*\*  
 \** ***@Route****("/login", name="admin\_login")  
 \** ***@return*** *Response  
 \*/***public function** admin\_loginAction() {  
 **return** $this->render('default/generic-login.html.twig',  
 [  
 'action' => $this->generateUrl('admin\_login\_check'),  
 'title' => 'Login des administrateurs'  
 ]);  
}

# Sécurité-Deconnexion

* Modification du security.yml, modifier la clé firewalls, admin\_firewall en ajoutant logout :

**logout:  
 path:** admin\_logout *# Cette route est gérée par Symfony, il suffit juste de l'a déclarer dans routing* **target:** homepage *# C’est la route en cas de succes*

* Modification de routing :

**admin\_logout:  
 path:** admin/logout

## Cryptage de mot de passe

Modifier security.yml, la clé « encoders » :

**encoders:  
 Symfony\Component\Security\Core\User\User:  
 algorithm:** bcrypt  
 **cost:** 15

puis taper la commande :

* php bin/console security:encode-password

Cette commande va me générer le mot de passe ‘123’ crypté.

Mettre le cryptage du mot de passe généré dans la clé password dans providers. Exemple :

**providers:  
 in\_memory:  
 memory:  
 users:  
 admin:  
 password:** $2y$15$6pckuGeTMRuWclhF.4VVuuCnvrgAXjLRfsJxAG1Y/FAbB0mjaCCqi  
 **roles:** ROLE\_ADMIN

## Récupérer login et error

Dans le controller,

$securityUtils = $this->get('security.authentication\_utils');  
  
$lastUserName = $securityUtils->getLastUsername();  
$error = $securityUtils->getLastAuthenticationError();

$lastUserName et $error doivent être affichés dans le twig de l’authentification

Traduction des messages :

* dans config.yml, mofifier la clé « locale » en « fr » et décommenter « translator » dans « framework »
* Créer un fichier « message.fr.yml » dans le dossier « translations »

Gestion autorisations

On crée un autre user  dans security.yml:

**mopao:** {**password:** $2y$15$GkEkVu3XkpAHwDew6wuLcuLCx.3Nr6WHYQhS90utewtQtYnGkj9PG, **roles:** ["ROLE\_SUPER\_ADMIN", "ROLE\_ADMIN"]}

On ajoute une autre clé dans sécurity.yml, au même niveau que la clé « firewalls »

**access-control:** - {**path:** ^/admin/secure, **roles:** ROLE\_SUPER\_ADMIN}

path: c’est le chemin sécurisé, roles: Qui a le droit d’y accéder

Dans le controller, faire:

*/\*\*  
 \** ***@Route****("/secure", name="admin\_only\_mopao")  
 \** ***@return*** *Response  
 \*/***public function** onlyMopaoAction() {  
 **return** $this->render('admin/mopao.html.twig', []);  
}

Remarque : pour faire les routes spécifiques, voir la doc

public function indexAction()

{

$user = $this->getUser()

$this->denyAccessUnlessGranted(

'ROLE\_ADMIN',

$user,

'Accès interdit'

);

}

## Hiérarchie des rôles :

Le super\_admin a aussi le rôle de allowed\_to\_switch

**role\_hierarchy:  
 ROLE\_SUPER\_ADMIN:** ROLE\_ALLOWED\_TO\_SWITCH  
*# ROLE\_SUPER\_ADMIN: [ROLE\_ALLOWED\_TO\_SWITCH]  
# ou  
# ROLE\_SUPER\_ADMIN:  
# - ROLE\_ALLOWED\_TO\_SWITCH*

# ENTITY USER (sécurité)

Utiliser une entité comme source de données pour faire de la sécurité. Le provider (la clé dans security.yml) devient notre entité

Dans notre entité, implementer 2 interfaces pour que Symfony le considère comme source de données

Exemple dans notre cas : Entity Author

**class** Author **implements** \Serializable, UserInterface

implémenter les méthodes (voir code) :

**public function** serialize()

**public function** unserialize($serialized)

**public function** getRoles()

**public function** getSalt()

**public function** getUsername()

Ajouter avec le getter et le setter

*/\*\*  
 \** ***@var*** *string  
 \*/***private** $plainPassword;

Une fois que c’est fait, générer un formulaire et remplir les attributs (voir code):

php bin/console doctrine:generate:form AppBundle:Author

Créer le controller et l’action. Dans notre cas et pour les soucis de simplification, on a tout mis dans DefaultController

Important : faire la commande suivante car on a créé une nouvelle entité :

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

On crée un nouveau provider dans security.yml

*# on crée un nouveau provider de donnée***author\_provider:  
 entity:** *# la source de donnée de ce provider est une entité* **class:** AppBundle:Author  
 **property:** email

Modifier le clé « encoders »

**encoders:  
 Symfony\Component\Security\Core\User\User:  
 algorithm:** bcrypt  
 **cost:** 15  
 **AppBundle\Entity\Author:** *# on associe un encoder à cette Entité* **algorithm:** bcrypt  
 **cost:** 15

Modifier la clé « main » comme suit :

**main:***# anonymous: ~* **anonymous:** true  
 **pattern:** ^/  
 **provider:** author\_provider  
 **form\_login:  
 login\_path:** author\_login  
 **check\_path:** author\_login\_check  
 **default\_target\_path:** homepage  
 **logout:  
 path:** author\_logout  
 **target:** homepage

modifier le routing.yml comme dans le cas de “Admin”. Voir plus haut

06/09/2017

Uploader une image/fichier

Il faut utiliser le bundle : stof-doctrine extension

* Modifier le fichier config.yml

**stof\_doctrine\_extensions:  
 default\_locale:** fr\_FR  
 **uploadable:** *# Options de "uploadable", on définit un chemin par défaut de téléchargement* **default\_file\_path:** "%kernel.root\_dir%/../web/uploadedFiles"  
 **orm:  
 default:  
 sluggable:** true *# On active l'option de Slug* **uploadable:** true *# On active l'option de téléchargement* **loggable:** true *# On active l'option loggable qui va activer une table d'historique de modification*

* Dans les annotations de la class qui représente notre entité, ajouter cette annotation :

@Gedmo\Uploadable avec des options. Mettre le « use » du namespace de Gedmo

Exemple :

*/\*\*  
 \* Post  
 \*  
 \** ***@ORM\Table****(name="posts")  
 \** ***@ORM\Entity****(repositoryClass="AppBundle\Repository\PostRepository")  
 \** ***@Gedmo\Uploadable****(  
 \* allowOverwrite=true,  
 \* filenameGenerator="SHA1",  
 \* maxSize="2000000",  
 \* allowedTypes="image/jpeg,image/png"  
 \* )  
 \*/***class** Post

{

….

}

* Ajouter dans la class, une propriété.

*/\*\*  
 \** ***@var*** *string  
 \** ***@ORM\Column****(name="image\_file\_name", type="string", length=80, nullable=true)  
 \** ***@Gedmo\UploadableFileName****()  
 \*/***private** $imageFilename;

* Créer des getters et setters

Enfin faire une maj de la base:

php bin/console doctrine:schema:update –force

* Ajouter un champs dans PostType.php
* ->add('imageFilename', FileType::*class*, ['label' => 'Image', 'required' => **false**])

Pour enregistrer le fichier, dans le controller, faire:

$uploadManager = $this->get('stof\_doctrine\_extensions.uploadable.manager');  
$uploadManager->markEntityToUpload($post, $post->getImageFilename());

ATTENTION: il faut active l’extention: « extension=php\_fileinfo.dll » dans php.info

Utilisation de « loggable » pour l’historique

Créer une class dans le dossier Entity qui extends AbstractLogEntry

Voir l’exemple :

**namespace** AppBundle\Entity;  
  
  
**use** Gedmo\Loggable\Entity\MappedSuperclass\AbstractLogEntry;  
**use** Doctrine\ORM\Mapping **as** ORM;  
  
*/\*\*  
 \** ***@ORM\Table****(name="logs")  
 \** ***@ORM\Entity****(repositoryClass="Gedmo\Loggable\Entity\Repository\LogEntryRepository")  
 \* Class Log  
 \** ***@package*** *AppBundle\Entity  
 \*/***class** Log **extends** AbstractLogEntry  
{  
  
}

Dans les annotations de Post.php

Ajouter :

***@Gedmo\Loggable****(logEntryClass="AppBundle\Entity\Log")*

Et Après on définit les champs qu’on veut suivre. Exemple titre et texte, on ajoute les notations :

***@Gedmo\Versioned****()*

Remarque : pour revenir à une version : on recupère le repository de Log. Exemple :

$repository = $this->getDoctrine()  
 ->getRepository("AppBundle:Log");

$logs = $repository->findAll();

….

$repository->revert($post, 2) // (l’entité, la version)