Doctrine et ORM 3

Requête SQL personnalisé

Parfois on a besoin de requête spécifique qui ne peuvent pas être fait avec les méthodes "find".

Pour cela on va faire appel au queryBuilder de Doctrine.
Rendons nous dans notre "**MessageRepository**".

```
/**
 * Selectionne les messages entre deux dates.
 *
 * @param string $min date de début
 * @param string $max date de fin
 * @return Message[] Retourne un tableau d'objet Message
 */
public function findByDateInterval(string $min, string $max): array
{
  return $this->createQueryBuilder('m')
              ->andWhere('m.createdAt BETWEEN :min AND :max')
              ->setParameter('min', $min)
              ->setParameter('max', $max)
              ->orderBy('m.createdAt', 'DESC')
              ->getQuery()
              ->getResult();
}
```

Ici on dit que l'on crée un nouveau **QueryBuilder** dont la table sera appelé "m".
Ici, on indique une clause WHERE avec un intervalle de dates, puis on injecte les valeurs via des paramètres nommés. Doctrine sécurise automatiquement la requête (pas d'injection SQL), mais attention à fournir des chaînes valides au format de date compatible avec la base (ex: 'YYYY-MM-DD'). Enfin un orderBy puis récupérer la requête et sur la requête on va récupérer les résultats.

Si je souhaite tester, il ne me reste qu'à commenter ces deux lignes dans l'affichage de mes messages et les remplacer par ces nouvelles :

```
# Requête Classique :
// $messages = $repo->findBy([], ["createdAt"=>"DESC"], $nb, ($page-1)*$nb);
// $total = $repo->count([]);
# Test requête personnalisé :
$messages = $repo->findByDateInterval("2023-06-01", "2024-06-01");
$total = count($messages);
```

Si on a plusieurs requêtes personnalisées qui utilisent les mêmes paramètres, par exemple si on a plusieurs variantes de notre "interval". Il est bon de factoriser cela.

Faire une fonction qui prendra le queryBuilder en paramètre et lui ajouterons les paramètres en commun, puis rendra le queryBuilder.

Les relations entre entités

Quel sont les trois types de relations ?

- One to One.
- One to Many.
- Many to Many.

En sql nous avions à créer des clefs étrangères (voir des tables associative pour le Many to Many) et en php nous devions faire des jointures pour récupérer nos informations.

Symfony nous simplifie grandement la tâche.

- Créons une nouvelle entité "**Category**" contenant un **name**(string 50 not null).
- Symfony va nous avoir créé notre nouvelle entité.
- Ensuite on va modifier notre entité "**message**", puis lui donner un champ "**category**", si on lui dit "**?**"; On pourra voir qu'on a le choix entre les différentes relations possible. Mais cela peut parfois être compliqué de savoir laquelle utiliser, On choisit “**relation**”.
    - il nous demande alors à quelle entité il sera lié, ici "**Category**".
    - Il nous détail alors les différentes possibilités. Ici, notre Message appartient à une Category, donc on sélectionne ManyToOne.
    - Puis il demande si cette propriété peut être null, dans notre cas, **Oui**;
    - Puis il nous demande si on souhaite ajouter une propriété à "**Category**" qui nous permettra d'accèder aux messages, On peut lui dire **oui**.
    - Il nous demande alors le nom de cette propriété, par défaut c'est le nom de notre entité en cours de création.

Nous avons terminé notre nouvelle entité, il ne nous reste plus qu'à suivre les instructions pour faire la nouvelle migration.

Une fois le "**make:migration**" fait. Quand vous allez tenter de faire le "**doctrine:migrations:migrate**" Dans certains cas, il peut y avoir une erreur. En effet si on tente d'ajouter une relation qui ne peut pas être **null** alors qu'une table contient déjà des entrées, cela provoquera une erreur car ces entrées précédentes n'auront pas de relation.

Pour arranger cela, plusieurs solutions, si vous êtes en développement avec aucune entrée importante, vous pouvez purger la bdd :

```
symfony console doctrine:database:drop --force
symfony console doctrine:database:create
symfony console doctrine:migrations:migrate
```

On a supprimé notre BDD, puis on l'a recréé et on a effectué nos migrations. Évidement, cette façon de faire fonctionne avec une BDD en cours de développement, n'allez pas supprimer toute les données d'un client.

On pourra juste supprimer les entrées de notre table. Ou bien alors accepter que les entrées soit **NULL**, puis corriger les anciennes entrées avant de modifier à nouveau en **NOT NULL**.

Il va nous rester à améliorer nos fixtures :

```
// Dans les uses :
use App\Entity\Category;
// modification de la méthode load :
public function load(ObjectManager $manager): void
{
  $faker = Factory::create();
  // Ajout de catégories :
  $categories = [];
  for($i=0; $i < 10; $i++)
  {
    $Category = new Category();
    $Category->setName($faker->word());
    $manager->persist($Category);
    $categories[] = $Category;
  }
  // Fin ajout catégorie.
  for ($i=0; $i < 30; $i++)
  {
    $Message = new Message();
    $Message  ->setContent($faker->realTextBetween(50,500))
            ->setCreatedAt(\DateTimeImmutable::createFromMutable($faker-
>dateTimeThisDecade()))
            // Ajout d'une catégorie à nos messages :
            // 90% qu'il donne un élément aléatoire et 10% de NULL
            ->setCategory($faker->optional(0.9)->randomElement($categories));

    $manager->persist($Message);
  }
  $manager->flush();
}
```

Une fois relancé on se retrouve avec 30 Messages et 10 Catégories.

Les jointures avec Symfony

Une fois qu'une relation est faite entre plusieurs de nos entités, si on a bien dit à Symfony de créer des propriétés correspondante dans nos entités, faire des jointures est très simple :

- Rendons nous dans notre template "message/index.html.twig"
- Ajoutons un bouton catégorie à nos messages à côté du bouton "delete".

```
{% if mess.category %}
  <a href="">
    {{ mess.category.name }}
```

```
    </a>
{% endif %}
```

Symfony comprend directement qu'il doit faire appel à la table "Category" depuis "Message" et qu'il doit afficher le "name" de "Category".

Si on fait un dump avant notre boucle, nous verrons que les informations dans "Category" sont null. Pourquoi? Car Symfony fait ce qu'on appelle du "**lazy loading**".

Si on n'a pas besoin d'une information, Symfony ne va pas la chercher. Il sait que les entités sont lié mais ne récupère que le stricte minimum.

Alors que si on fait le dump à la fin, les informations seront visible.

```
{{ dump(messages[0]) }}
```

Doctrine utilise le **lazy loading** : les relations ne sont chargées que lorsqu'on y accède. C'est pour cela que *mess.category* est *null* au début du template. Dès qu'on accède à *category.name*, Doctrine fait automatiquement une requête SQL pour récupérer la catégorie associée.

Les évènement doctrine "lifecycle"

Doctrine lance certains évènement lors du cycle de vie d'une entité, lorsque vous le créez, le mettez à jour ou alors le supprimez.

- PrePersist
- PostPersist
- PreUpdate
- PostUpdate
- PreRemove
- PostRemove

On peut ajouter à nos entités des fonctions lancé durant ces évènements. Prenons notre entité "**Message**".

```
// juste au dessus de la classe:
#[ORM\HasLifecycleCallbacks()]
// Dans ma classe entité (par exemple en bas):
#[ORM\PreUpdate()]
public function onPreUpdate()
{
  $this->editedAt = new \DateTime();
}
```

Maintenant, si je tente d'éditer un Message, son "**editedAt**" sera automatiquement mis à jour.

Mais quand on y pense, on pourrait faire de même sur le createdAt et cela sur toute nos entités qui méritent un createdAt et un editedAt... Or un développeur n'aime pas se répéter, c'est donc ici qu'interviennent les

traits.

Lifecycle et trait

Créons un dossier et un fichier "**src/Traits/TimeStampTrait.php**".

```
namespace App\Traits;

use Doctrine\ORM\Mapping as ORM;

trait TimeStampTrait
{}
```

On va retirer les propriétés "**createdAt**" et "**EditedAt**" de notre Message et les placer dans le trait.
On va faire de même avec les setters et getters.
Ainsi que le lifecycle que l'on vient de créer.

```
#[ORM\Column]
private ?\DateTimeImmutable $createdAt = null;

#[ORM\Column(nullable: true)]
private ?\DateTime $editedAt = null;
public function getCreatedAt(): ?\DateTimeImmutable
{
  return $this->createdAt;
}

public function setCreatedAt(\DateTimeImmutable $createdAt): static
{
  $this->createdAt = $createdAt;

  return $this;
}

public function getEditedAt(): ?\DateTime
{
  return $this->editedAt;
}

public function setEditedAt(?\DateTime $editedAt): static
{
  $this->editedAt = $editedAt;

  return $this;
}

#[ORM\PreUpdate()]
public function onPreUpdate()
{
```

```
    $this->editedAt = new \DateTime();
}
```

Ajoutons pendant qu'on y est le lifecycle suivant :

```
#[ORM\PrePersist()]
public function onPrePersist()
{
    $this->createdAt = new \DateTimeImmutable();
}
```

Il ne nous reste plus qu'à ajouter le "**use**" dans nos classes "**Message**" et "**Category**";

```
// en haut de la page :
use App\Traits\TimeStampTrait;
// au dessus de la classe:
#[ORM\HasLifecycleCallbacks()]
// dans la classe:
use TimeStampTrait;
```

Plus qu'à faire les migrations(il vous faudra peut être supprimer les données et refaire les fixtures).

Rappel :

```
# Supprimer la bdd
symfony console doctrine:database:drop -f
# Créer la bdd
symfony console doctrine:database:create
# Relancer les migrations :
symfony console doctrine:migrations:migrate
# Relancer les fixtures :
symfony console doctrine:fixtures:load
```

Maintenant plus besoin de créer à chaque nouvelle entité ces deux champs, on aura juste à ajouter ce trait.

Le trait peut être réutilisé dans toutes les entités qui ont besoin d'un createdAt et editedAt. On évite ainsi de répéter du code, ce qui respecte le principe DRY (Don't Repeat Yourself).