Doctrine et ORM 2

Persistance des données et Repository

Maintenant que notre BDD est créé, nous allons faire un nouveau controller qui va nous permettre de tester les fonctionnalités de Doctrine.

Enregistrer des données

Appelons le "**MessageController**"

```
# Symfony/Cli est fait pour trouver lui même la commande qui correspond le mieux à votre demande :
symfony console m:cont
# En indiquant juste cela il va comprendre qu'on veut faire un "make:controller"
# Si il a un doute sur votre demande, il indiquera que votre commande est "ambigous" et vous proposera celles avec lesquelles il hésite.
```

Puis je vais appliquer quelques modifications à mon controller :

```
// Ajouter avec les autres "use"
use App\Entity\Message;
use Doctrine\ORM\EntityManagerInterface;

// Pour que toute mes routes commencent par "/message", ajoutez au dessus de ma classe :
#[Route('/message')]

// Puis je vais modifier ma méthode "index" pour qu'elle ressemble à cela :
#[Route('/add', name: 'app_message_create')]
public function createMessage(EntityManagerInterface $em): Response
{
    $message = new Message();
    $message    ->setContent("Ceci est un message de test")
                ->setCreatedAt(new \DateTimeImmutable());

    $em->persist($message);
    $em->flush();
    return $this->render('Message/index.html.twig', [
        'controller_name' => 'MessageController',
    ]);
}
```

Que retrouvons nous ici :

- Je récupère dans mon paramètre "$em" un objet "**Entity Manager**" qui me permettra de manipuler mes entités.

- J'instancie un nouveau Message.
- Je remplis les propriétés de mon Message.
- Puis je lui dit de "**persist**" (Je lui indique qu'il va devoir être enregistré en base de données)
- Et une fois terminé je "**flush**" (Valider toute les actions demandé précédemment)

Sans le flush, ce que j'ai pu faire avant, ne sera pas enregistré.
Avec lui on peut voir que les requêtes sql apparaissent dans la barre de Symfony.

Je peux très bien faire toute une liste d'action avant de flush, ajoutons juste avant le flush() :

```
$message2 = new Message();
$message2   ->setContent("Ceci est un second message de test")
            ->setCreatedAt(new \DateTimeImmutable());

$em->persist($message2);
```

persist plusieurs éléments avant de faire un flush permet de regrouper toute les opérations en une seule transaction, et ainsi gagner en efficacité.

Si l'on va sur notre page *http://127.0.0.1:8000/message/add* ou qu'on l'actualise, on ajoutera donc nos messages en BDD.

On verra dans l'outil de suivi de Doctrine que les actions sont enregistré mais envoyé ensemble plutôt que une à une. Evidement dans un cas normal, nous récupérons les informations venant d'un formulaire et non écrite à la main comme cela.

Créer des fixtures

Lorsque l'on veut tester un site, on a souvent besoin de tester nos fonctionnalités, mais créer un à un toute les données que l'on pourrait avoir à supprimer ou refaire est long et fastidieux.
C'est là qu'interviennent les fixtures. Ce sont des données de test que l'on va pouvoir charger avec une simple commande. Pour cela installons ceci :

```
# Bibliothèque officielle géré par symfony :
composer require --dev orm-fixtures
# Simple Bibliothèque PHP:
composer require --dev fakerphp/faker
```

Le "--dev" indique que ce bundle ne doit être utilisé qu'en environnement de développement.
Quand à fakerphp, lui il va permettre de générer aléatoirement des données. Vous trouverez la documentation ici : https://fakerphp.github.io/

Symfony a par défaut créé un fichier "AppFixtures.php" mais histoire de bien ranger nos fixtures, nous allons créer le notre. Lançons donc la commande suivante :

```
symfony console make:fixture
```

Et indiquons lui que l'on veut créer "**MessageFixtures**". Notons que ajouter "Fixtures" à la fin n'est pas obligatoire mais permettra de s'y retrouver plus facilement dans nos noms de fichier. Rendons nous dans ce fichier qui se trouve dans "**src/DataFixtures/** et recopions le code suivant :

```
// Avec les autres "use" :
use Faker\Factory;
use App\Entity\Message;

// Dans la méthode "load" :
$faker = Factory::create();
for ($i=0; $i < 30; $i++)
{
    $message = new Message();
    $message  ->setContent($faker->realTextBetween(50,500))
            ->setCreatedAt(\DateTimeImmutable::createFromMutable($faker-
>dateTimeThisDecade()));

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

- Dans la méthode "**load**" on crée un nouveau "**factory**" venant de Faker, objet auquel on pourra demander de générer des informations aléatoire.
- Puis on fait une boucle d'autant de message que l'on souhaite créer.
- On demande à faker de rentrer des informations aléatoires (texte et date, voir la documentation de faker pour plus de fonctionnalités).
    - Notons que "**\DateTimeImmutable::createFromMutable**" est necessaire ici car faker crée un "DateTime" classique mais que notre entité attend un "DateTimeImmutable".
- On persiste, puis une fois la boucle terminée, on flush.

Une fois notre fixtures prête, on n'a plus qu'à lancer la commande suivante :

```
symfony console doctrine:fixtures:load
```

Symfony va nous demander si est sûr de faire cela car cela va vider la BDD. Dans notre cas où elle est vide, cela ne nous dérange pas. Mais si jamais vous voulez juste ajouter de nouvelles données sans effacer les précédentes, vous pouvez ajouter l'option "**--append**" à votre ligne de commande.

Si jamais vous poussez cela plus loin, il est possible de load seulement certains fichiers fixtures en créant des groupes puis demandant de charger seulement un groupe en particulier et autres options pratique.

Repository

On a vu comment ajouter des éléments, maintenant voyons comment récupérer les données de notre BDD. Pour cela on va devoir récupérer le repository, nous avons deux façons de faire cela, ajoutons cette méthode à notre "**MessageController.php**" :

Tous récupérer

```
// Avec les autres "use" :
use App\Repository\MessageRepository;

// Dans la classe :
#[Route('/', name: 'app_message_read')]
public function readMessage(MessageRepository $repo): Response
{
    $messages = $repo->findAll();
    return $this->render('message/index.html.twig', [
        'messages' => $messages,
    ]);
}
```

Ici on profite de l'autowiring pour récupérer le MessageRepository.

Ensuite on utilise sur notre repository la méthode "**findAll()**" qui comme son nom l'indique permet de récupérer toute notre table.

On notera qu'il existe différentes façon de récupérer le repository.

Maintenant rendons nous dans notre template "**message/index.html.twig**" et changeons tout le contenu du block body :

```
{% for mess in messages %}
  <div class="message">
      <div class="date1">
          Ajouté le {{mess.createdAt | date("d/m/Y") }}
      </div>
      <div class="date2">
          {{mess.editedAt ? "édité le : " ~ mess.editedAt|date("d/m/Y")  : ""}}
      </div>
      <p>{{ mess.content }}</p>
      <div class="btns">
        {# Nous ajouterons les différents boutons plus tard #}
      </div>
  </div>
{% endfor %}
```

Et voilà, nous avons un tableau contenant toutes nos Messages.
Les repository contiennent tout un tas de méthode permettant les requêtes SQL les plus communes, et bien évidement, si on a besoin de requête plus complexe, nous pouvons créer les nôtres.

Si nous regardons le fichier "**MessageRepository**" Nous verrons que des methodes commenté qui servent d'exemple à des constructions de requête SQL.

Mais les repositories héritent de leur parent plusieurs methodes par défaut :

- "**findAll()**" pour tout récupérer.
- "**find()**" qui récupère la ligne correspondant à l'id donné en paramètre.
- "**findOneBy()**" qui permet en lui donnant un tableau associatif de trouver le premier résultat correspondant aux paramètres données.
- "**findBy**" qui fonctionne comme le précédent mais pour récupérer tous les résultats correspondant.

Un exemple d'utilisation serait par exemple de remplacer notre **findAll()** par un **findBy()** pour trier par date de création.

```
$messages = $repo->findBy([], ["createdAt"=>"DESC"]);
```

Le premier paramètre correspond au `WHERE` et le second au `ORDER BY`.

Pagination

La pagination en symfony peut être géré par des bibliothèques externe supplémentaires, mais évidement on va le faire nous même.

Allons sur notre methode readMessage :

```
// On modifie la route :
#[Route('/{page<\d+>?1}/{nb<\d+>?5}', name: 'app_message_read')]
// On ajoute les paramètres $nb et $page
public function readMessage(MessageRepository $repo,$nb, $page): Response
{
  // On ajoute les paramètres 3 et 4 à findBy.
  $messages = $repo->findBy([], ["createdAt"=>"DESC"], $nb, ($page-1)*$nb);
  // On compte le total d'élément en BDD.
  $total = $repo->count([]);
  // On calcul le nombre de page
  $nbPage = ceil($total / $nb);
  // On envoi nos nouveaux paramètres à la vue.
  return $this->render('Message/index.html.twig', [
    'messages' => $messages,
    "nbPage" => $nbPage,
    "nombre" => $nb,
    "currentPage"=>$page
  ]);
}
```

- On récupère dans l'adresse, le numéro de la page et combien d'élément on souhaite par page. (avec une valeur par défaut si rien n'est fourni)
- On utilise findBy, mais cette fois on lui donne le nombre de résultat que l'on souhaite par page et à partir du quel nombre chercher. (LIMIT et OFFSET)
- On récupère le nombre total de message.
- On calcul le nombre de page maximum.
- et on envoie tout cela a une nouvelle vue.

Pour le template, j'ai gardé notre affichage mais j'ai ajouté là où je souhaite que mes pages s'affichent :

```
<div class="pagination">
    {% if currentPage > 1 %}
        <a href="{{ path("app_message_read", {nb:nombre, page:1}) }}">Première</a>
|
        <a href="{{ path("app_message_read", {nb:nombre, page:currentPage - 1})
}}">Précédent</a> |
    {% endif %}
    {% for i in range(max(1, currentPage - 2), min(nbPage, currentPage + 2)) %}
        <a href="{{ path("app_message_read", {nb:nombre, page:i}) }}" style="font-
weight:{{i==currentPage?"bold":"normal"}}">{{i}}</a> |
    {% endfor %}
    {% if currentPage < nbPage %}
        <a href="{{ path("app_message_read", {nb:nombre, page:currentPage + 1})
}}">Suivant</a> |
        <a href="{{ path("app_message_read", {nb:nombre, page:nbPage})
}}">Dernière</a>
    {% endif %}
</div>
```

Et nous voilà grâce à ce code avec une pagination fonctionnelle, évidement on peut améliorer le design.

Récupérer une ligne par id

Nous avons récupéré toute notre BDD, mais parfois nous ne souhaitons récupéré qu'une seule ligne de notre bdd, classiquement tentons de le faire via l'id. Pour cela créons une nouvelle route :

```
#[Route('/delete/{id<\d+>}', name: 'app_message_delete')]
public function deleteMessage(MessageRepository $repo, $id): Response
{
    $message = $repo->find($id);

    if($message)
    {
        $this->addFlash("info", "TODO: Supprimer message");
    }
    else
    {
        $this->addFlash("error", "Aucun message correspondant");
    }
    return $this->redirectToRoute("app_message_read");
}
```

Basiquement on récupère l'id dans le paramètre de la route, et on utilise la méthode "**find()**" pour trouver la Message dont c'est l'id.

Si on trouve le message, on le supprime (ici juste un message flash) sinon on indique que la suppression n'a pas pu être faite. Puis on redirige notre utilisateur ailleurs.

Supprimer un élément

Pour supprimer un élément, ce n'est pas plus compliqué, voici mon controller :

```
#[Route('/delete/{id<\d+>}', name: 'app_message_delete')]
public function deleteMessage(MessageRepository $repo, EntityManagerInterface $em,
$id): Response
{
  $message = $repo->find($id);

  if(!$message)
  {
      $this->addFlash("error", "Aucun message correspondant");
  }
  else
  {
    // On demande la suppression
    $em->remove($message);
    // On valide toute nos demandes
    $em->flush();
    // On change le message flash
    $this->addFlash("info", "Message supprimé");
  }
  return $this->redirectToRoute("app_message_read");
}
```

On récupère l'entity manager. Depuis l'entity manager, on fais appel à la méthode "**remove**" en lui donnant l'entité que je souhaite supprimer, puis je **flush()** pour valider la demande.

On ajoute un flash message pour confirmer ou infirmer la suppression.

Côté twig, on peut rajouter un lien dans notre affichage des messages dans la div boutons :

```
<a href="{{ path("app_message_delete", {id: mess.id})}}">supprimer</a>
```

Il nous manque encore à vérifier si nos utilisateurs sont connecté et si ils sont le propriétaire du message devant être supprimé, mais on verra cela quand on aura des utilisateurs.

Mettre à jour une entité

Nos liens fonctionnent, c'est bien, mais en vérité, nous avons écrit trop de code. Récupéré une entrée de la BDD selon son ID est vraiment quelque chose de très commun, c'est pour cela que Symfony est capable de le gérer bien plus facilement grâce au **param converter**.

Testons cela sur une nouvelle méthode:

```
#[Route('/update/{id<\d+>}', name: 'app_message_update')]
public function updateMessage(?Message $message): Response
{
  if($message)
  {
    $this->addFlash("info", "TODO update message");
  }
  else
  {
    $this->addFlash("error", "Aucun message correspondant");
  }
  return $this->redirectToRoute("app_message_read");
}
```

Ici Symfony voit qu'on lui fourni un paramètre, et qu'on demande en premier argument de nous trouver un objet. Il va alors automatiquement faire l'association et trouver l'élément dont c'est l'id. Le "?" indique que l'élément peut être "NULL", c'est important car si on indique un ID ne correspondant à rien, Symfony va remplir la variable avec la valeur "NULL".

Tout ce qu'on a fait précédemment est géré automatiquement ici.

Normalement un update se ferait avec un formulaire, mais on a pas encore vu les formulaires. Donc pour cet exemple on passera les informations en paramètre de route.

```
// On ajoute à notre route le contenu du message
#[Route('/update/{id<\d+>}/{content}', name: 'app_message_update')]
// On récupère l'entity manager et le contenu
public function updateMessage(?Message $message, EntityManagerInterface $em,
string $content): Response
{
  if($message)
  {
    // On modifie notre message
    $message->setContent($content)
            ->setEditedAt(new \DateTime());
    // On sauvegarde notre modification de message
    $em->flush();
    $this->addFlash("info", "Message mis à jour");
  }
  else
  {
    $this->addFlash("error", "Aucun message correspondant");
  }
  return $this->redirectToRoute("app_message_read");
}
```

On remarque que la mise à jour est quasiment identique à la création : la seule différence est qu'au lieu d'instancier un nouvel objet avec new Message, on récupère un message existant, et qu'il n'est pas nécessaire d'appeler persist() sur cet objet, car Doctrine suit déjà cette entité en mémoire.

Pour l'instant notre create et notre update fonctionnent selon les informations qu'ils ont en paramètre de route ou directement écrit dans le code. Il nous reste quelques détails à voir à propos des requêtes et des entités mais après cela nous verrons comment fonctionnent les formulaires.