Doctrine et ORM

Pour discuter avec la base de donnée, Symfony utilise la bibliothèque **Doctrine**. Cette dernière est ce qu'on appelle un ORM (*Object Relation Mapper*). Son rôle est de faire le lien entre la base de donnée et les objets que l'on crée dans notre code.

Pour prendre un exemple, si on souhaite créer une table "article" alors on va créer une classe "article" et Doctrine lira cette classe et créera la table correspondante.

Mais avant tout, on a besoin d'indiquer à symfony où se trouve notre BDD, pour cela rendons nous dans le fichier "**.env**".
(*Pour rappel, ici on travail sur xampp avec aucune information secrète. Donc envoyer cela sur git n'est pas un problème, mais lorsque vous travaillez avec de vrai mot de passes et une vrai BDD, créez un fichier* "***.env.local***" *qui lui ne sera pas envoyé sur git.*)

Trouvez les lignes correspondantes à doctrine et tapez :

```
DATABASE_URL="mysql://root:@127.0.0.1:3306/blog-symfony?serverVersion=10.4.18-MariaDB&charset=utf8mb4"
```

- "mysql://" pour le driver.
- "root:" pour le nom d'utilisateur.
- "" cet espace vide est le mot de passe.
- "@127.0.0.1:3306/" l'adresse et le port de la BDD.
- "blog-symfony" le nom de la bdd.
- "?serverVersion=10.4.18-MariaDB" la version de la BDD.
- "&charset=utf8mb4" l'encodage utilisé.

Une fois cela validé, vous pouvez taper dans le terminal :

```
symfony console doctrine:database:create
```

Symfony vous aura créé lui même la BDD.

Entity

La première étape pour faire usage de Doctrine, cela va être de créer cette première classe, imaginons qu'on souhaite une table contenant des villes. Et bien on pourrait créer la classe à la main, mais évidement, Symfony va nous mâcher le travail.

On avait vu le "make:controller", maintenant c'est au tour de :

```
symfony console make:entity
```

Symfony va ensuite nous demander plusieurs choses :

- Un nom pour l'entité (essayez de toujours mettre une majuscule). Symfony nous pose d'autres questions auxquels on pourrait répondre, mais on y reviendra plus tard. appuyons sur **entrée** pour mettre fin à cela.

On voit qu'il a créé deux fichiers, laissons pour l'instant de côté le repository et regardons l'entité.

- Cela commence évidement par un namespace.
- Puis le use du repository qui lui correspond.
- Et le use de doctrine en tant que ORM.
- Au dessus de la classe, on trouve un attribut indiquant la liaison de la classe avec le repository du même nom.
- Dans notre classe nous trouvons par défaut une propriété id et son getter. Cet id se voit attribué 3 lignes indiquant que c'est un ID, qu'il doit être généré et que c'est une colonne.
  On remarquera aussi que cet id est de type int ou null.

Maintenant qu'on a vu cela, retapons exactement la même commande que précédemment, le "**make:entity**" et indiquons lui encore une fois Ville.
Symfony va voir que cette entité existe déjà et va nous proposer de lui ajouter des propriétés :

- Ajoutons lui "**nom**"
- Il va nous demander le type de cette propriété, en tapant "?" vous pourrez voir tous les types disponible. Mais par défaut, on le voit entre crochet, si on ne lui indique rien, il prendra le type **string** qui nous convient parfaitement.
- Les questions suivantes peuvent varier selon le type choisi, ici il nous demande la taille de notre string, je vais indiquer **50**.
- Ensuite il nous demande si ce champ peut être null avec par défaut "no", parfait, je ne le souhaite pas null.
- Nous revoilà à la case départ, il nous redemande si on veut ajouter une autre propriété ou alors s'arrêter.
- Ajoutons un champ "**population"** integer non null.
- Puis un champ "**createdAt**"... Mais que voyons nous, il nous propose par défaut un champ "datetime_immutable". parfois symfony reconnaît certains noms de propriété et propose le type adapté. Validons cela avec un not null.
- Enfin ajoutons un "**editedAt**", mais changeons ce "datetime_immutable" par un "datetime" classique. et indiquons lui qu'il peut être null.

Retournons voir notre classe :

- Toutes nos propriétés ont été ajouté.
- Elles sont typé et conditionné.
- Elles sont toute accompagné d'un setter et d'un getter.

La prochaine étape va être d'envoyer cette classe en BDD.

Migrations

Maintenant que nous avons créé notre entité, il ne nous reste plus qu'à créer la "**migration**", c'est un fichier qui va contenir toute les requêtes SQL pour convertir votre entité en table.

Pour cela, si vous regardez votre terminal, symfony vous a déjà indiqué quoi faire :

"Next: When you're ready, create a migration with php bin/console make:migration"

à noter que "php bin/console" est la même chose que "symfony console".

```
symfony console make:migration
```

Symfony a comparé la BDD actuelle avec ses fichiers et a créé une migration qui correspond aux changements remarqués.
Il nous demande maintenant de vérifier le fichier puis de lancer une nouvelle commande.

Obéissons lui. On va trouver dans la migration 3 méthodes :

- getDescription, qui fait ce qu'elle dit.
- up qui contient les requêtes SQL à lancer pour mettre à jour notre BDD.
- down qui permettra d'inverser cette migration si jamais on a un problème.
- On notera que cette première migration crée aussi une seconde table liée à une bibliothèque de Symfony, mais on ne s'en occupera pas.

Si cela nous convient, on peut envoyer la migration :

```
symfony console doctrine:migrations:migrate
```

il nous indique que l'on va changer notre BDD et que si on n'a pas fait attention, cela pourrait nous faire perdre des données, mais on souhaite continuer alors validons cela.

Si on regarde notre BDD, on verra notre nouvelle table définie. Ainsi qu'une table qui sauvegarde quels sont les migrations que l'on a déjà faite. Pour qu'il ne relance pas toutes les migrations à chaque fois.

Si je lui relance la fonction maintenant, il me dira qu'il n'y a rien à changer.

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 "**VilleController**"

Je vais d'ailleurs préfixer ses routes d'un "**/ville**".
Puis créer la méthode suivante :

```
// en haut
use App\Entity\Ville;
use Doctrine\Persistence\ManagerRegistry;
```

```php
// Dans la classe
#[Route('/add', name: 'addVille')]
public function create(ManagerRegistry $doc): Response
{
    $em = $doc->getManager();
    $ville = new Ville();
    $ville  ->setNom("Lille")
            ->setPopulation(234475)
            ->setCreatedAt(new \DateTimeImmutable());

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

Que retrouvons nous ici :

- ManagerRegistry va nous permettre d'enregistrer des éléments en BDD.
- Je récupère dans ma variable "$em" mon "**Entity Manager**"
- J'instancie une nouvelle ville.
- Je remplie les propriétés de ma ville.
- Puis je lui dit de "**persist**" (Je lui indique qu'elle va devoir être enregistré en BDD)
- 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() :

```php
$ville2 = new Ville();
$ville2  ->setNom("Nice")
        ->setPopulation(342669)
        ->setCreatedAt(new \DateTimeImmutable());

$em->persist($ville2);
```

On verra dans l'outil de suivi de Doctrine que les actions sont enregistré mais envoyé ensemble plutôt que une à une.

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 :

```
composer require --dev doctrine/doctrine-fixtures-bundle
composer require 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/

Maintenant, lançons la commande suivante :

```
symfony console make:fixture
```

Et indiquons lui que l'on veut créer "**VilleFixtures**". Rendons nous dans ce fichier qui se trouve dans "**src/DataFixtures/** et recopions le code suivant :

```
// En haut
use Faker\Factory;
// Dans la méthode "load"
$faker = Factory::create();
for ($i=0; $i < 10; $i++)
{
    $ville = new Ville();
    $ville  ->setNom($faker->city())
            ->setPopulation($faker->randomNumber(6, true))
            ->setCreatedAt(\DateTimeImmutable::createFromMutable($faker-
>dateTimeThisDecade()));

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

- Dans la méthode "**load**" on crée un nouveau "**factory**" venant de Faker.
- Puis on fait une boucle d'autant d'entré que l'on souhaite.
- On demande à faker de rentrer des informations aléatoires.
- 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, 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 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 "**VilleController.php**" :

Tous récupérer

```
#[Route('/', name: 'readVille')]
public function read(ManagerRegistry $doc): Response
{
    $repo = $doc->getRepository(Ville::class);
    $villes = $repo->findAll();
    return $this->render('ville/index.html.twig', [
        'villes' => $villes,
    ]);
}
```

Ici on demande à notre ManagerRegistry de nous récupérer le repository qui est lié à la classe Ville.

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

Si on souhaite se passer du ManagerRegistry on peut aussi écrire :

```
use App\Repository\VilleRepository;
public function read(VilleRepository $repo): Response
```

Maintenant rendons nous dans notre template et changeons un peu tout cela :

```
<table>
    <thead>
        <tr>
            <th>id</th>
            <th>nom</th>
            <th>action</th>
        </tr>
    </thead>
    <tbody>
    {% for v in villes %}
        <tr>
            <td>{{v.id}}</td>
            <td>{{v.nom}}</td>
            <td><a href="">Voir</a></td>
```

```
        </tr>
    {% endfor %}
    </tbody>
</table>
```

Et voilà, nous avons un tableau contenant toutes nos villes.
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 "**VilleRepository**" Nous verrons certaines des méthodes disponible :

- "**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.

Si ces 4 là sont en commentaire de la documentation de la classe, c'est car elles ne sont pas défini ici mais dans les classes hérités.

Puis les méthodes "**add()**" et "**remove()**" assez simple à comprendre.

Et enfin en commentaire, des exemples de comment créer vos propres méthodes pour créer vos requêtes **SQL** avec Symfony.

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('/{id<\d+>}', name: 'detailVille')]
public function detail(ManagerRegistry $doc, $id): Response
{
    $repo = $doc->getRepository(Ville::class);
    $ville = $repo->find($id);
    if(!$ville) return $this->redirectToRoute("readVille");
    return $this->render('ville/detail.html.twig', [
        'ville' => $ville,
    ]);
}
```

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

Si on ne trouve aucune ville, on redirige ailleurs, on pourrait d'ailleurs ajouter des flash messages.

Côté twig on va modifier les liens de notre liste de ville et créer **detail.html.twig** :

```
<!-- dans notre liste de ville -->
{{ path("detailVille", {id: v.id}) }}
<!-- Dans le détail de la ville -->
<h1>{{ville.nom | upper}}</h1>

<p>
  {{ville.nom}} a une population de {{ ville.population }} habitants.
  <br>
  Cette fiche a été créé le {{ville.createdAt | date}}.
  {% if ville.editedAt %}
    <br>
    Cette fiche a été édité le {{ville.editedAt | date}}.
  {% endif %}
</p>
```

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**.

Remplaçons notre ancienne méthode détail par :

```
#[Route('/{id<\d+>}', name: 'detailVille')]
public function detail(Ville $ville=null): Response
{
    if(!$ville) return $this->redirectToRoute("readVille");
    return $this->render('ville/detail.html.twig', [
        'ville' => $ville,
    ]);
}
```

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.

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

Récupérer des lignes par colonnes

Il est aussi possible de préciser que l'on souhaite rechercher une ligne par autre chose que son id, pour cela on utilisera "**findBy**" et "**findOneBy**" :

```
#[Route('/{name}/{nb?1}', name: 'readVilleName')]
public function readByName(VilleRepository $repo,$nb, $name): Response
{
  if($nb > 1)
  {
    $villes = $repo->findBy(["nom"=>$name], [], $nb);
    return $this->render('ville/index.html.twig', [
        'villes' => $villes,
```

```
        ]);
    }
    $ville = $repo->findOneBy(["nom"=>$name]);
    return $this->render('ville/detail.html.twig', [
        'ville' => $ville,
    ]);
}
```

Pour l'exemple j'appelle directement le repository, mais j'aurais pu faire comme avant et utiliser l'entity manager.

Ici je tente de récupérer ma ville par son nom. le second paramètre ici en tableau vide est pour faire un **orderBy**, le troisième est le nombre de résultat maximum que l'on souhaite, et le quatrième que je n'utilise pas ici est un **offset**, indiquant à partir de quel résultat récupérer.

Pagination

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

Il aurait été bien de le faire sur notre page "readVille" mais histoire de ne pas perdre les exemples précédents, on va créer une page spécifique à cela :

```
#[Route('/page/{page?1}/{nb?5}', name: 'readVillePage')]
public function readPaginate(VilleRepository $repo,$nb, $page): Response
{
    $villes = $repo->findBy([], [], $nb, ($page-1)*$nb);
    $total = $repo->count([]);
    $nbPage = ceil($total / $nb);
    return $this->render('ville/pagination.html.twig', [
        'villes' => $villes,
        "nbPage" => $nbPage,
        "nombre" => $nb,
        "page"=>$page
    ]);
}
```

- On récupère dans l'adresse, le numéro de la page et combien d'élément on souhaite par page.
- On utilise findBy sans lui préciser les deux premiers paramètres, mais on lui donne le nombre que l'on souhaite par page et à partir du quel chercher.
- On récupère le nombre total de Ville.
- On calcul le nombre de page maximum.
- et on envoie tout cela a une nouvelle vue.

Pour le template, j'ai repris le même tableau que pour la page "readVille" auquel j'ai ajouté :

```
<div class="pagination">
    {% if page > 1 %}
```

```
    <a href="{{ path("readVillePage", {nb:nombre, page:1}) }}">précédent</a> |
    {% endif %}
    {% for i in range(1, nbPage) %}
        <a href="{{ path("readVillePage", {nb:nombre, page:i}) }}">{{i}}</a> |
    {% endfor %}
    {% if page < nbPage %}
    <a href="{{ path("readVillePage", {nb:nombre, page:nbPage}) }}">suivant</a> |
    {% endif %}
</div>
```

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

Supprimer un élément

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

```
#[Route('/delete/{id<\d+>}', name: 'deleteVille')]
public function delete(Ville $ville=null, ManagerRegistry $doc): Response
{
  if($ville)
  {
    $em = $doc->getManager();
    $em->remove($ville);
    $em->flush();
  }
  return $this->redirectToRoute("readVille");
}
```

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

On pourrait ajouter des flashes messages pour confirmer ou infirmer la suppression.

Côté twig, on peut rajouter un lien dans notre colonne action à côté de "voir" :

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

Mettre à jour une entité

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.

```
#[Route('/update/{id<\d+>}/{nom}/{pop<\d+>}', name: 'updateVille')]
public function update(Ville $ville=null, ManagerRegistry $doc, $nom, $pop):
Response
{
```

```
  if($ville)
  {
    $ville  ->setNom($nom)
            ->setPopulation($pop);
    $em = $doc->getManager();
    $em->persist($ville);
    $em->flush();
  }
  return $this->redirectToRoute("readVille");
}
```

On remarquera que l'update est exactement pareil que le create à la différence que au lieu de faire un "**new Ville**" on récupère une ville existante.

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 "**VilleRepository**".

```
/**
 * @return Ville[] Returns an array of Ville objects
 */
public function findByPopulationInterval($min, $max): array
{
  return $this->createQueryBuilder('v')
    ->andWhere('v.population BETWEEN :min AND :max')
    ->setParameter('min', $min)
    ->setParameter('max', $max)
// ->setParameters(["min"=>$min, "max"=>$max])
    ->orderBy('v.population', 'ASC')
    ->getQuery()
    ->getResult()
  ;
}
```

Ici on dit que l'on crée un nouveau **QueryBuilder** dont la table sera appelé "v".
On va lui indiquer un **WHERE** puis on va lui donner ses paramètres.
Enfin un orderBy puis récupérer la requête et sur la requête on va récupérer les résultats.

Il ne nous reste plus qu'à faire notre route dans le controller :

```
#[Route('/interval/{min}/{max}', name: 'intervalVille')]
public function interval(VilleRepository $repo, $min, $max): Response
{
  $villes = $repo->findByPopulationInterval($min, $max);
  return $this->render('ville/index.html.twig', [
      'villes' => $villes,
```

```
    ]);
}
```

Ma nouvelle route utilise le même twig que précédemment mais au lieu de faire un **findAll**, elle utilise notre nouvelle méthode.

Si on a plusieurs requêtes personnalisés 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 "**departement**" contenant un **nom**(string 50 not null). et un **code**(string 5 not null).
- Ensuite on va lui donner un "**chefLieu**", 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, donc disons lui "**relation**"
    - il nous demande alors à quelle entité il sera lié, ici "**Ville**".
    - Il nous détail alors les différentes possibilités. Dans notre cas c'est le OneToOne.
    - Puis il demande si cette propriété peut être null, dans notre cas, **Oui**;
    - Puis il nous demande si on souhaite ajouter une propriété à "**Ville**" qui nous permettra d'accèder au département, 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, mais nous allons l'appeler "chefLieu".
- Ensuite ajoutons lui une propriété "**Villes**" qui sera aussi une relation.
    - Elle sera liée à **Ville** aussi.
    - On sera là sur un OneToMany.
    - Il nous indique ce coup ci qu'on n'a pas le choix et qu'une nouvelle propriété va être ajouté à "Ville". Mais il nous demande le nom. (ici departement).
    - Peut-elle être Null? disons lui que Non.
    - Ensuite il nous demande si on souhaite supprimer les villes orphelines, donc si un département disparaît, doit on supprimer les villes. On va lui dire que Non.

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

Mais une fois le "**make:migration**" fait. Quand vous allez tenter de faire le "**doctrine:migrations:migrate**" vous allez tomber sur une erreur. En effet les villes que l'on a créé avant, ne contenait pas de département, et on lui a dit que ça ne pouvait pas être Null.

Pour arranger cela, on va faire :

```
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.

Il va nous rester à améliorer nos fixtures :

```
$faker = Factory::create();
        for ($j=0; $j < 10; $j++) {
            $dep = new Departement();
            $dep->setNom($faker->state())
                ->setCode($faker->randomNumber(2, true));
            for ($i=0; $i < 10; $i++)
            {
                $ville = new Ville();
                $ville  ->setNom($faker->city())
                        ->setPopulation($faker->randomNumber(6, true))
                        -
>setCreatedAt(\DateTimeImmutable::createFromMutable($faker->dateTimeThisDecade()))
                        ->setDepartement($dep);

                $manager->persist($ville);
            }
            $dep->setChefLieu($ville);
            $manager->persist($dep);
        }
        $manager->flush();
```

Une fois relancé on se retrouve avec 100 villes et 10 départements.

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 :

- Ajoutons une colonne département à notre tableau de ville.
- Et dans la boucle pour la case département écrivons :

```
{{v.departement.nom}}
```

Symfony comprend directement qu'il doit faire appel à la table "departement" depuis "ville" et qu'il doit afficher le "nom" du département.

Si on fait un dump au début de notre boucle, nous verrons que les informations dans "departement" 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.

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 nos "**Villes**".

```
// juste au dessus de la classe en dessous de l'attribut :
#[ORM\HasLifecycleCallbacks()]
// en bas de mon entité:
#[ORM\PreUpdate()]
public function onPreUpdate()
{
  $this->editedAt = new \DateTime();
}
```

Maintenant, si je tente d'éditer une ville, 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;
use Doctrine\DBAL\Types\Types;
```

```
trait TimeStampTrait
{}
```

On va retirer les propriétés "**createdAt**" et "**EditedAt**" de notre Ville 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.

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 "**Ville**" et "**Departement**";

```
// 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 sûrement supprimer les données et refaire les fixtures).

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