*Developpez votre site web avec le framework Symfony*

**Les services**

* Appel d’un service : **« $this->get(‘serviceName’); »**
* Les services doivent être déclarés dans **« XBundle/Ressources/config/services.yml »**
* Doctrine et l’EntityManager sont des services.

**La couche métier : les entités**

La couche métier est représentée par l’ORM (Object Relation Mapper). Le rôle de ce dernier et de s’occuper de la base de données… Sans qu’on s’en rende compte ! Il utilise en effet des objets et des fonctions plutôt que des requêtes SQL. Un objet confié à l’ORM s’appelle plus précisément une « entité » et enregistrer cette entité signifie la « persister ».

Le service qui gère réellement les données de la BDD est en fait l’EntityManager mais ce dernier doit être instancié après Doctrine. C’est lui, en effet, qui exécute les requêtes SQL.

Pour récupérer les données ce sont par contre les repositories qui sont à l’ouvrage. Ce sont des objets qui utilisent l’EntityManager. Il existe un repository par entité.

L’EntityManager MANIPULE les données alors que les repositories RECUPERENT les données.

**Manipuler ses entités**

* Doctrine est un service, ce qui signifie qu’il remplit une fonction bien précise et est accessible partout dans notre code. La fonction de Doctrine est de gérer la persistance de nos objets, autrement dit la connexion à la base de données (partie DBAL) ainsi que la gestion des entités (partie ORM) grâce à l’EntityManager.

Récupérer doctrine :**« $doctrine = $this->getDoctrine(); »**

* EntityManager est également un service. C’est en réalité ce dernier qui s’occupe de tout… Sauf de la récupération des entités qui elle se fera avec les Repositories.

Récupérer l’EntityManager : **« $em = $doctrine->getManager(); »**

* Les Repositories sont des objets qui utilisent l’EntityManager et servent à la récupération des entités. Il existe un Repository par entité mais plusieurs entités peuvent avoir le même Repository et aucune requête SQL ne doit être faite en dehors d’un Repository.

Récupérer un Repository : «**$RepositoryX = $em->getRepository(‘Bundle:X’);**»

* Persister un objet n’exécute aucune requête SQL mais cela indique à Doctrine que c’est maintenant lui qui gère le dit objet. Il faut utiliser le flush pour enregistrer concrètement les données, c’est la validation par transaction :
  1. Création d’entité : **« $x = new X ; »**
  2. Persistance de l’entité : **« $em -> persist($x); »**
  3. Envois de l’entité en BDD : **« $em -> flush($x); »**

**Récupérer ses entités avec Doctrine 2**

Il existe deux manières de récupérer ses entités depuis un Repository :

* Le QueryBuilder qui construit des requêtes SQL par étapes.
* Le DQL (Doctrine Query Language) qui n’est qu’une adaptation du SQL adapté à la vision objet de Symfony.

Utilisation du QueryBuilder :

* Le QueryBuilder permet de construire nos Queries, mais il n’est pas lui-même une Query. Il se récupère grâce à l’EntityManager et plus exactement depuis une méthode d’un Repository :

**« $qb = $this -> \_em -> createQueryBuilder(‘a’); »**

* Il existe 4 méthodes dites « normales » pour récupérer ses entités fournies directement par Doctrine :
  1. **« find($id) »** récupère dans une instance de l’objet l’entité d’un Repository en fonction de son id
  2. **« findAll() »** récupère dans un tableau d’objets toutes les entités contenues dans la BDD
  3. **« findBy() »** récupère dans un tableau d’objets une liste d’entités auxquelles on peut appliquer des filtres (where, etc…)
  4. **« findOneBy() »** récupère dans une instance de l’objet l’entité d’un Repository en fonction de filtres (where, etc…)
* S’ajoute à ces méthodes normales deux méthodes dites « magiques » fournies directement par PHP :
  1. **« findByX($val) »** en remplaçant X par une propriété de notre entité (ex : « findByTitle($val); »). La méthode fonctionne comme une findBy() avec pour seul filtre X.
  2. **« findOneByX($val) »** en remplaçant X par une propriété de notre entité (ex : « findOneByAuthor($val); ». La méthode fonctionne comme un findOneBy avec pour seul filtre X.
* Le résultat obtenu par le QueryBuilder est récupéré par une Query pour traitement avec extraction :

**« $query = $qb -> getQuery(); »**

* Enfin on récupère le résultat dans une variable :
  1. **« $result = $query -> getResult(); »** : retourne le resultat sous forme d’un tableau d’objet sur lequel on pourra faire des modifications.
  2. **« $result = $query -> getArrayResult(); »**: retourne le resultat sous forme d’un tableau de tableaux que lequel on ne pourra pas faire de modifications mais qui s’avère être plus rapide en lecture que le « getResult(); »
  3. **« $result = $query -> getScalarResult(); »**: retourne un tableau de valeurs, par exemple pour un COUNT.
  4. **« $result = $query -> getSingleScalarResult(); »**: retourne la seule valeur du résultat, ou une erreur si plus d’une valeur.
  5. **« $result = $query -> getSingleResult(); »**: : retourne le résultat s’il est unique ou le premier résultat s’il y’en a plusieurs.
  6. **« $result = $query -> getOneOrNullResult(); »**: retourne le résultat s’il est unique, le premier résultat s’il y’en a plusieurs, ou null s’il n‘y en a pas.
  7. **« $result = $query -> execute(); »**: utilisé pour exécuter des requêtes qui ne retournent pas de résultat complexes (update, insert, etc...)

Utilisation du DQL :

* Le DQL s’utilise directement pour créer une Query. Les méthodes pour ensuite récupérer le résultat restent les mêmes :

**« $query = $this -> \_em -> createQuery(‘requête’); »**

* La requête DQL se construit comme une requête SQL mais on utilise le nom des entités à la place de celui des tables. Il est également obligatoire de donner un alias à l’entité cible qui permettra de sélectionner tous les attributs de la dite entité sous la forme d’objet(s) modifiable(s), l’équivalent de l’étoile \* en SQL :

**« SELECT x FROM Bundle:Entity x »**

* Il est possible de tester ses requêtes DQL directement depuis la console :

**«  php bin/console doctrine:query:dql ‘’requête’’ »**

Utiliser les jointures dans nos requêtes :

* Une jointure permet de récupérer avec une entité A toutes ses entités B liées. La jointure n’est possible que si l’entité propriétaire contient un attribut pointant vers l’un de ceux de l’entité jointe.
* Pour faire une jointure :

1. **« $qb = $this -> createQueryBuilder(‘x’) »** : On sélectionne l’entité propriétaire.
2. **« -> innerJoin(‘x.attribut’, ‘y’) »** : On indique l’attribut qui fait la jointure et on donne un alias à l’entité jointe.
3. **« ->addSeect(‘y’); »** : On sélectionne l’entité jointe, pas besoin de précisé laquelle car Doctrine la retrouve grâce aux annotations.
4. « **-> innerJoin(‘x.attribut’, ‘y’, ‘WITH’, ‘YEAR(x.attribut) = val’)**» : On peut rajouter une condition à la jointure.

**Les évènements et extension Doctrine**

Les évènements Doctrine :

Les évènements, ou « callbacks », d’une entité sont des méthodes que Doctrine va exécuter pour nous à certains moments précis, selon le cycle de vie, ou « lifecycle », de l’entité.

* Avant de créer un callback il faut préciser à Doctrine que notre entité en contiendra, sans quoi ils seront tout simplement ignorés. Cela se fait grâce à l’annotation « HasLifecycleCallbacks » qui faut rajouter dans la déclaration de l’entité cible :

**« @ORM\HasLifecycleCallbacks() »**.

* Il faut ensuite liée la méthode à son évènement toujours en utilisant les annotations et en choisissant selon le moment où l’on veut que l’action s’effectue parmi les 7 possibilités suivantes :

**« @ORM\event »**

* 1. PrePersist
  2. PostPersist
  3. PreUpdate
  4. PostUpdate
  5. PreRemove
  6. PostRemove
  7. PostLoad
* Le problème avec les évènements qu’on définit directement dans les entités c’est qu’ils n’ont justement accès qu’aux informations de leur propre entité. La solution est de définir l’événement non plus dans une entité mais dans un service.

**/!\ Attention** : Avec cette méthode le service est exécuté après un évènement peu importe l’entité, il faut donc tester le type de l’entité si on ne veut pas effectuer une action commune à toutes nos entités.

**/!\ Attention** : La méthode crée dans le service doit porter le même nom que l’évènement (postPersist, preUpdate, etc…).

Les extensions Doctrine :

* Symfony disposant d’une large communauté, de nombreuses extensions existent déjà permettant de faire bon nombre de tâches liées aux entités.
* StofDoctrineExtensionBundle intègre déjà différentes extensions Donctrine bien utilies. Pour l’installer il faut modifier le fichier composer.json avant d’executer un composer update.