Base de Données et Doctrine

Qu'est-ce qu'un ORM ?

Un **ORM** (Object-Relational Mapping) est un outil qui permet de **mapper** (c'est-à-dire de faire correspondre) des **objets** de votre code à des **tables** d'une base de données **relationnelle**. En d'autres termes, il crée un pont entre le monde de la programmation orientée objet (où vous manipulez des objets) et le monde des bases de données relationnelles (où les données sont structurées en tables).

Pourquoi utiliser un ORM ?

- **Abstraction:** Il vous permet de vous concentrer sur la logique métier en vous affranchissant des détails de la base de données (requêtes SQL complexes, jointures, etc.).
- **Productivité:** Il automatise de nombreuses tâches répétitives comme la création des tables, la gestion des relations entre les données, etc.
- **Qualité du code:** Il favorise un code plus propre et plus maintenable en séparant les préoccupations.

Doctrine : l'ORM de Symfony

Doctrine est un ORM très populaire, notamment dans l'écosystème Symfony. Il offre de nombreuses fonctionnalités avancées :

- **Mapping objet-relationnel :** Doctrine utilise des annotations pour définir la correspondance entre les propriétés d'une classe et les colonnes d'une table.
- **DQL (Doctrine Query Language):** Ce langage de requête permet de formuler des requêtes complexes de manière orientée objet, en utilisant des objets plutôt que du SQL brut.
- **Relations:** Doctrine gère facilement les relations entre les entités (One-to-One, One-to-Many, Many-to-Many).
- **Migrations:** Doctrine permet de gérer les évolutions de votre schéma de base de données de manière déclarative, en générant automatiquement les scripts de migration.

- **Validation:** Doctrine intègre un système de validation pour s'assurer de l'intégrité des données.
- **Évènements:** Doctrine permet de s'abonner à des événements du cycle de vie des entités (avant la persistance, après la suppression, etc.).

En résumé, Doctrine simplifie considérablement l'interaction avec une base de données dans un contexte Symfony. Il permet aux développeurs de se concentrer sur la logique métier plutôt que sur les détails d'implémentation de la base de données.

Etape 1 : connecter votre base de donnée au projet

```
DATABASE_URL="mysql://identifiant:password@127.0.0.1:3306/base_name?serverVersion=8.0.32&charset=utf8mb4”
```

Vous pouvez creer votre base de donnée dans php my admin et la connecter a votre projet mais vous pouvez aussi la créer depuis le terminal après avoir renseigné le DATABASE_URL dans le .env en utilisant la commande suivante

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

Etape 2 : Ajouter des tables

Pour cela vous utiliserez les commandes suivantes :

1️⃣ créer une entité

```
symfony console make:entity
```

Le terminal vous posera différentes questions pour nommer la table, ajouter les champs avec leur types

2️⃣ Créer un fichier migration

```
symfony console doctrine:make:migration
```

👀 ouvrez le fichier nouvellement crée et regarder ce qu'il contient , vous reconnaissez ici des requetes SQL que nous allons pouvoir executer maintenant

3️⃣ Effectuer la migration : exécuter les requêtes SQL et création des fichier entité et repository associé

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

⇒ remarquez : la table a été ajoutée dans votre base de donnée , 2 fichiers sont apparus : l'entité et le repository correspondant ces 2 fichiers correspondent a ce que vous apellez "model" jusqu'a présent, regardons les de plus près

Entity et Repository

- L'entité représente les colonnes de la bdd, les attributs de notre objet avec des information sur le format et la façon dont l'ORM mappe les informations, elle contient aussi les différents getters/ setters
- Le repository quand a lui contient le méthodes de communication avec la base de donnée, il dispose de méthodes préconstruites comme find($id) pour aller récupérer une donnée par son id, findAll(), findBy().

```
/**
 * @method Product|null find($id, $lockMode = null, $lockVe
rsion = null)
 * @method Product|null findOneBy(array $criteria, array $o
rderBy = null)
 * @method Product[]    findAll()
 * @method Product[]    findBy(array $criteria, array $orde
rBy = null, $limit = null, $offset = null)
 */
```

- Dans le repository vous implémenterez les méthodes d’interaction avec la BDD spécifiques dont vous pouvez avoir besoin dans votre code.

  exemple une méthode qui va chercher uniquement les produits disponibles le la catégorie “électroménager” en les triant par prix croissant , en utilisant le querybuilder de doctrine :

```
public function findAvailableProductsInElectromenagerCatego
ry(): array
    {
        return $this->createQueryBuilder('p')
            ->where('p.category = :category')
            ->andWhere('p.available = :available')
            ->setParameter('category', 'électroménager')
            ->setParameter('available', true)
            ->orderBy('p.price', 'ASC')
            ->getQuery()
            ->getResult()
        ;
    }
```

Les relations

Doctrine utilise quatre types de relations :

1. **One-to-One:** C'est comme un mariage où chaque personne n'est mariée qu'à une seule autre personne. Un utilisateur peut n'avoir qu'un seul profil, par exemple.
2. **One-to-Many:** C'est comme une famille où un parent a plusieurs enfants, mais un enfant n'a qu'un seul parent. Un auteur peut avoir plusieurs livres, mais un livre n'a qu'un seul auteur.
3. **Many-to-One:** C'est l'inverse de la relation précédente. Plusieurs éléments sont liés à un seul élément. Par exemple, plusieurs produits peuvent appartenir à une seule catégorie.
4. **Many-to-Many:** C'est comme des amis : une personne peut avoir plusieurs amis, et ces amis peuvent avoir plusieurs autres amis. Un utilisateur peut avoir plusieurs rôles, et un rôle peut être attribué à plusieurs utilisateurs.

En résumé:

Doctrine vous permet de définir ces relations entre vos objets de manière simple et intuitive. Grâce à ces relations, vous pouvez créer des structures de données complexes et naviguer facilement entre elles.

> 💡 Commencez par créer les tables vers lesquelles pointent les liaisons, pour la relation Many-To-Many symfony se chargera tout seul de gerer la table intermédiaire.

Exemple concret:

Imaginez un site de vente en ligne. Vous pourriez avoir les entités suivantes :

- **Produit:** Chaque produit a un nom, un prix et appartient à une catégorie.
- **Catégorie:** Une catégorie regroupe plusieurs produits.
- **Commande:** Une commande contient plusieurs produits.
- **Client:** Un client peut passer plusieurs commandes.

Dans ce cas, vous auriez les relations suivantes :

- **Produit** et **Catégorie** : Une relation Many-to-One (plusieurs produits vers une catégorie).
- **Commande** et **Produit** : Une relation Many-to-Many (une commande peut contenir plusieurs produits, et un produit peut être dans plusieurs commandes).
- **Client** et **Commande** : Une relation One-to-Many (un client peut passer plusieurs commandes, mais une commande n'appartient qu'à un seul client).

En utilisant Doctrine, vous pourriez facilement récupérer tous les produits d'une catégorie donnée, ou tous les produits commandés par un client particulier.

Comment utiliser les entities et les repositories dans un Controller

Rien de plus simple ! grâce a l’injection dépendances !

```
class ProductController extends AbstractController{
    #[Route("/product/list", name='product_list' , methods=
['GET'])]
    public function list(ProductRepository $ProductReposito
ry){
        $products = $ProductRepository->findAll()
        ...
    }
}
```

Enregistrer des données dans la table (create / update)

Pour cela il vous faudra injecter la dépendance **EntityManager**

Pour la création vous instanciez un objet (entité) vide, vous hydratez votre objet (donner des valeurs aux attributs avec les setters) puis utiliser 2 méthodes de l'entityManager qui sont persist et flush

La méthode `persist`

- **Rôle:** Elle marque une entité comme étant à persister. En d'autres termes, elle indique à Doctrine qu'elle doit prendre en compte cette entité lors de la prochaine opération de `flush`.
- **Fonctionnement:** Lorsque vous appelez `persist` sur une entité, Doctrine la place dans son unité de travail (UnitOfWork). Cette unité de travail garde une trace des modifications apportées aux entités et des nouvelles entités à créer.

La méthode `flush`

- **Rôle:** Elle déclenche la synchronisation entre l'unité de travail et la base de données.
- **Fonctionnement:** Lorsque vous appelez `flush`, Doctrine analyse les changements apportés aux entités depuis le dernier `flush` et génère les requêtes SQL nécessaires pour les appliquer à la base de données. Ces

requêtes peuvent être des insertions, des mises à jour ou des suppressions.

Exemple :

```
class UserController extends AbstractController{
    #[Route("/user/add", name="user_add", methods=["POS
T"])]
    public function add(EntityManager $entityManager){
        // Création d'un nouvel utilisateur
        $user = new User();
        $user->setUsername('johnDoe');
        $user->setEmail('johndoe@example.com');

        // Persistance de l utilisateur
        $entityManager->persist($user);

        // Exécution des requêtes SQL
        $entityManager->flush();
    }
}
```

Exercice

A partir de l’exercice que nous avons fait en front (dashboard) et que vous avez réutilisé en php

créer les tables categories et produits ainsi que les champs associés les champs associés .

Nous allons prévoir 2 controllers :

- un app controller qui gerera les routes du site public (accessible par tout le monde)
- un controller pour les produits que l’on sécurisera plus tard pour n’être accessible que par l’ADMIN

dans le AppController ainsi que les méthodes de ce controller (et les templates) permettant de :

1️⃣ Voir la liste des produits

2️⃣ au click sur ‘voir’ , afficher un produit par son id

> 👀 Par la suite nous allons devoir parler des formulaires avant de pouvoir persiter des information provenant d’une saisie utilisateur