👩‍💻 Doctrine

Author Muanbeno Francis 👋🏾

DOCTRINE ORM

Object relational mapping

Doctrine est une bibliothèque utilisée pour gérer les bases de données dans les projets PHP.

Il permet de mapper les tables de la base de données, de générer des requêtes en utilisant un langage simple.

Cela facilite la gestion des données dans un projet, ce qui permet de se concentrer sur la logique de l'application plutôt que sur les détails de la gestion de la base de données.

En résumé, Doctrine est un outil qui facilite la communication entre votre application PHP et votre base de données.

☐ 1 - Créer un dossier “crud” dans votre serveur local (htdoc) pour Mamp ou www pour Xamp/Wamp.

☐ 2 - Ouvrez le dossier “crud” avec votre IDE (visual studio code) puis créer un fichier “composer.json” et un dossier “src” à la racine de votre dossier

☐ 3 - Insérer le code ci-dessous dans le fichier “composer.json”

```
{
  "require": {
      "doctrine/orm": "*",
      "doctrine/dbal": "*",
      "symfony/yaml": "^6.2",
      "symfony/cache": "^6.2"
    },
  "autoload": {
      "psr-0": {"": "src/"}
  }
}
```

DOCTRINE DBAL

database abstraction layer

***DBAL** est une couche d'abstraction pour les **bases de données**.*

*Il permet aux développeurs de travailler avec des bases de données de **différents types** (comme MySQL, PostgreSQL, etc.) en utilisant une seule interface commune.*

*Cela signifie que les développeurs peuvent écrire du code qui fonctionne avec **différents types de bases de données sans avoir à changer leur code**.*

☐ 4 - Exécuter la commande ci-dessous dans un terminal pour installer doctrine.

```
composer install
```

☐ 5 - Créer un fichier bootstrap dans lequel il faudra insérer le code ci-dessous

```
<?php
// bootstrap.php
use Doctrine\DBAL\DriverManager;
use Doctrine\ORM\EntityManager;
use Doctrine\ORM\ORMSetup;

require_once "vendor/autoload.php";
```

```
// Create a simple "default" Doctrine ORM configuration for Attributes
$config = ORMSetup::createAttributeMetadataConfiguration(
    paths: array(__DIR__."/src"),
    isDevMode: true,
    cache: null,
    proxyDir: null
);

// configuring the database connection
$connection = DriverManager::getConnection([
    'driver' => 'pdo_mysql',
    'host' => 'localhost', // localhost:8889 sur mamp
    'user' => 'root',
    'password' => '', // 'root' sur mamp
    'dbname' => 'boutique'
], $config);

// obtaining the entity manager
$entityManager = new EntityManager($connection, $config);
```

☐ 6 - Remplacer tout le contenu du fichier vendor/bin/doctrine par le code ci-dessous

```
#!/usr/bin/env php
<?php
// bin/doctrine

use Doctrine\ORM\Tools\Console\ConsoleRunner;
use Doctrine\ORM\Tools\Console\EntityManagerProvider\SingleManagerProvider;

// Adjust this path to your actual bootstrap.php
require __DIR__ . '../../../bootstrap.php';

ConsoleRunner::run(
    new SingleManagerProvider($entityManager)
);
```

☐ 7 - Exécuter la commande ci-dessous pour vérifier que la configuration à bien été effectuée.

```
php vendor/bin/doctrine orm:schema-tool:create
```

Comme nous n'avons pas encore ajouté **d'entité** dans src, vous verrez un message indiquant "Aucune classe de métadonnées à traiter". Dans la prochaine section, nous allons créer une entité “**Produit**” ainsi que les métadonnées correspondantes, et exécuter à nouveau cette commande.

☐ 8 - Maintenant nous allons créer notre première entity “Produit”, pour cela il faudra créer un fichier “produit.php” qui contiendra le code ci-dessous dans le dossier src.

```
<?php

use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity]
#[ORM\Table(name: 'produits')]
class Product

{
  #[ORM\Id]
  #[ORM\Column(type: 'integer')]
  #[ORM\GeneratedValue]
  private int|null $id = null;

  #[ORM\Column(type: 'string')]
  private string $name;

  // .. (other code)

  public function getName(): string
  {
    return $this->name;
  }

  public function setName(string $name): void
  {
    $this->name = $name;
  }

  public function getId(): int
  {
    return $this->id;
  }
}
```

☐ 9 - créer la basse de donnée “boutique” dans phpMyAdmin.

☐ 10 - Exécuter la commande ci-dessous pour créer la table produit dans la bdd que vous venez de créer.

```
php vendor/bin/doctrine orm:schema-tool:create
```

☐ 11 - Créer un fichier dans le dossier src “post_product.php” , ce fichier nous permettra d’insérer des produits en bdd.

☐ 12 - Exercice (POST) : Insérer 5 produits en bdd en utilisant la méthode persiste() & flush()

> La méthode **persist()** est utilisée pour indiquer à doctrine que l'objet doit être enregistré dans la base de données.
>
> **flush()** permet d’exécuter la requête généré par la method **persiste()**.

1 - Créez une variable qui contient le nom du produit, cela peut être un jean, un sac, etc.

2 - Instancier la classe Produit.

3 - Utilisez le setter pour mettre à jour le nom de votre produit.

4 - Utilisez la méthode persist() provenant de l'objet entityManager pour indiquer à Doctrine que l'objet doit être enregistré dans la base de données.

5 - Utilisez la méthode flush() provenant de l'objet entityManager pour exécuter votre persistance.

Correction : post_produit.php

En résumé, le code ci-dessous crée un nouvel objet de la classe Produit avec le nom "Basket" et le stocke dans la base de données en utilisant Doctrine.

```
<?php

require_once "../bootstrap.php";

$article = "Basket";

/*
  Cela crée une nouvelle instance de la classe
  Produit et stocke l'objet dans la variable $product.
```

```
*/
$product = new Produit;

/*
  Cela appelle la méthode setName() de l'objet $product
  et lui passe la valeur de la variable $article comme argument.
*/
$product->setName($article);

/*
  @persist()
  La méthode persist() est utilisée pour indiquer à Doctrine
  que l'objet doit être enregistré dans la base de données lors de
  l'appel à flush().
*/
$entityManager->persist($product);

/*
  @flush()
  Cela permet de confirmer toutes les modifications apportées à la
  base de données et de les enregistrer de manière permanente en utilisant
  la méthode flush() de l'entityManager.
*/
$entityManager->flush();
```

☐ 13 - Exercice (GET) : Créer un fichier “read.php” dans lequel il faudra récupérer tous les produits dans votre bdd et les afficher dans votre console.

Correction : get_produit.php

En résumé, le code ci-dessous récupère tous les objets de la classe Produit stockés dans la base de données en utilisant Doctrine, puis les affiche un par un avec leur identifiant et leur nom.

```
<?php

require_once '../bootstrap.php';

/*
  @$entityManager->getRepository()
  Cela crée une instance de l'objet repository
  pour la classe Produit et le stocke dans la variable $repoProduit.
  Le repository est une classe qui fournit des méthodes
  pour interagir avec les objets de la base de données.
*/
$repoProduit = $entityManager->getRepository(Produit::class);

/*
  Cela appelle la méthode findAll() du repository de l'objet Produit,
  pour récupérer tous les objets de la base de données et les stocke
  dans la variable $produits.
*/
$produits = $repoProduit->findAll();
```

```
/*
  Cela parcourt tous les objets stockés dans la variable $produits
  en utilisant une boucle foreach.
 */
foreach ($produits as $produit) {
  /*
  Cela affiche l'ID et le nom de chaque objet Produit stocké dans
  la base de données.
  La méthode getId() renvoie l'identifiant de l'objet
  La méthode getName() renvoie le nom de l'objet.
*/
  echo $produit->getId() . ' : ' . $produit->getName() . "\n";
}
```

☐ 14 - Exercice (GET) : Effectuer la même chose dans un fichier “read2.php” en utilisant la méthode **createQueryBuilder()**, ce qui vous permettra de personnaliser vos requêtes.

Correction : get_produit_builder.php

```
<?php

require_once '../bootstrap.php';

/*
  @createQueryBuilder
  Cette ligne crée un objet QueryBuilder à partir d'un objet EntityManager,
  qui permet de construire une requête.
*/
$query = $entityManager->createQueryBuilder()
          ->select('p.name', 'p.id') /*
            Cette ligne spécifie les colonnes que l'on souhaite récupérer
            dans la requête.
            Ici, on veut récupérer les colonnes "name" et "id" de la table
            "Produit", que l'on aliasse en "p".
          */
          ->from(Produit::class, 'p') /* Cette ligne spécifie la table sur
            laquelle la   requête sera exécutée. Ici, la table est "Produit"
            (une classe PHP qui est mappée à une table dans la base de données),
            et on utilise l'alias "p" pour y faire référence.
          */
          ->getQuery(); /*Cette ligne retourne la requête sous forme
            d'objet Query.*/

$produits = $query->getArrayResult(); /* Cette ligne exécute la requête
et retourne les résultats sous forme de tableau associatif.
Les résultats sont stockés dans la variable $produits. */
```

☐ 15 - Exercice (GET) : Récupérer le produit qui a l’id 2, vérifier que le produit existe, si le produit existe affichez son id et son name sinon affichez “produit not

found” .

Correction : get_produit_ById.php

```
<?php

require_once "../bootstrap.php";

$id = 2; /* Cette ligne définit la valeur de l'identifiant $id que l'on
souhaite récupérer. */

$product = $entityManager->find(Product::class, $id);/* Cette ligne récupère
un objet Product à partir de son identifiant $id en utilisant la méthode
find de l'EntityManager. L'identifiant est la clé primaire du produit. */

$productId = $product->getId(); /* Cette ligne récupère l'identifiant de
l'objet Product à partir de la méthode getId() et stocke cette valeur
dans la variable $productId.
 */

$entityManager->remove($product); /* Cette ligne supprime l'objet $product
de la base de données en utilisant la méthode remove de l'EntityManager. */
```

Correction : get_produit_ById_builder.php

```
<?php

require_once "../bootstrap.php";

$id = 1;
$query = $entityManager->createQueryBuilder()
->select('p.id', 'p.name')
->from(Produit::class, 'p')
->where('p.id = :id') /* Cette méthode ajoute la clause WHERE à la requête,
en spécifiant la condition de recherche. Dans cet exemple,
nous recherchons le produit qui a un identifiant égal à 1,
en utilisant une variable nommée :id. */
->setParameter('id', $id)/*  Cette méthode associe une valeur à la variable
nommée :id dans la requête, en utilisant la valeur de la variable
$id définie précédemment.
Cela permet d'éviter les attaques par injection SQL et de rendre
la requête dynamique.
 */
->getQuery();
$result = $query->execute();
```

16 - Exercice (DELETE) - Effectuer une suppression.

Correction : delete_produit.php

```php
<?php
/*
  On récupère l'objet "Produit" correspondant à l'identifiant $id
  en utilisant la méthode "find" de l'EntityManager
*/
$produit = $entityManager->find(Produit::class, $id);

// Si l'objet n'a pas été trouvé, $produit sera null
if ($produit !== null) {
    // On supprime l'objet de la base de données en utilisant
  // la méthode "remove" de l'EntityManager
    $entityManager->remove($produit);
    // On enregistre les modifications en base de données
    // en utilisant la méthode "flush" de l'EntityManager
    $entityManager->flush();
}
```

Correction : delete_produit_builder.php

```php
// On définit l'identifiant de l'objet "Produit" à supprimer
$id = 1;

$query = $entityManager->createQueryBuilder()
            ->delete(Produit::class, 'p')
            ->where('p.id = :id')
            ->setParameter('id', $id)
            ->getQuery();

$query->execute();
```

17 - Exercice (PUT) - Effectuer une modification.

Correction : put_produit.php

```php
// On définit l'identifiant de l'objet "Produit" à modifier
$id = 1;

// On définit le nouveau nom pour l'objet "Produit"
$newProduit = 'basket';

// On récupère l'objet "Produit" correspondant à l'identifiant $id en
// utilisant la méthode "find" de l'EntityManager
$produit2 = $entityManager->find(Produit::class, $id);

// Si l'objet a été trouvé, on modifie son nom en utilisant
// la méthode "setName"
if ($produit2 !== null) {
    $produit2->setName($newProduit);
    // On enregistre les modifications en base de données
    //en utilisant la méthode "flush" de l'EntityManager
```

```
    $entityManager->flush();
}
```

Correction : put_produit_builder.php

```
$id = 1;
$name = "Basket";

$produit3 = $entityManager->createQueryBuilder()
                    ->update(Produit::class,'p')
                    ->set('p.name',':name')
                    ->where('p.id = :id')
                    ->setParameter('name', $name)
                    ->setParameter("id",$id)
                    ->getQuery();

$produit3->execute();
```