COUCHE D'ACCÈS AUX DONNÉES

LA COUCHE D'ABSTRACTION PDO

PDO est une extension définissant l'interface qui permet de communiquer avec differents types de

bases de données. Avec PDO, il n'est pas nécessaire de changer les fonctions d'accès aux differents SGBD dans son code, mais seulement les paramètres du constructeur de la classe PDO.

COUCHE D'ACCÈS AUX DONNÉES

ACCÈS À LA BASE DE DONNÉES MYSQL AVEC PDO

```php
// Variables de connexion
$user = 'root';//utilisateur de la BD
$password = 'root'; //mdp de la BD
$dsn = 'mysql:host=localhost;dbname=stage_db';
try {

  $dbh = new PDO($dsn, $user, $password);
  // On définit le gestionnaire d'erreur
 $dbh->setAttribute(PDO::ATTR_ERRMODE,PDO::ERRMODE_EXCEPTION);
} catch (PDOException $e) {
  //capture de l'exception en cas d'erreur de connexion ou en
  echo "Erreur : " . $e->getMessage();
}
```

COUCHE D'ACCÈS AUX DONNÉES

REQUÊTE SELECTION AVEC LA MÉTHODE QUERY

```php
$sql = "SELECT title, description, name FROM post p JOIN autho
$sth = $dbh->query( $sql );
$data = $sth->fetch(PDO::FETCH_ASSOC); //PDO::FETCH_ASSOC recu
foreach( $data as $ $row ){
  echo $row['title'];
  echo $row['name'];

  echo $row['description'];
}
```

La méthode **query()** est utilisée pour toutes les requêtes en lecture seule comme SELECT, SHOW, DESC, EXPLAIN

COUCHE D'ACCÈS AUX DONNÉES

AJOUT DES DONNÉES AVEC LA MÉTHODE EXEC()

```
$sql = "INSERT INTO post(titlte, description, created_at, auth
VALUES('article #1', 'Lorem ipsum dolor sit amet.', NOW(), 1)"

$sth = $dbh->exec( $sql );
$result= $sth->fetch(PDO::FETCH_ASSOC); //PDO::FETCH_ASSOC rec
if( $result === FALSE )
   echo "Echec lors de l'insertion";
else
   echo "Vos données ont été ajoutées avec succès";
```

La méthode **exec()** est utilisée pour toutes requêtes de modification comme INSERT, UPDATE, DELETE

On utilise courramment les requêtes preparées quand on reçoit des données venant de l'exterieur sinon notre application vulnérable aux attaques par **Injection SQL**

COUCHE D'ACCÈS AUX DONNÉES

REQUÊTE PREPARÉE

Une requête preparée s'effectue en 2 étapes:
- La préparation: Un modèle de requête(sans les données) est envoyé au serveur. A ce stade le serveur effectue la vérification syntaxique, l'enregistre(le temps de l'exécution du script) pour une utilisation ulterieure.
- L'exécution: à ce stade le serveur exécute la requête avec les données reçues pour reconstituer la requête réelle.

Les requ preparées ont pour avantage:
- Amélioration des performations quand une même requête est appelée plusieurs fois.
- Protection des attaques par injection SQL car les paramètres des requêtes preparées sont automatiquement protegées par le pilote du SGBD.

COUCHE D'ACCÈS AUX DONNÉES

AJOUT DES DONNÉES AVEC DES PARAMÈTRES NOMMÉS

```php
$title = 'article #1';
$content = 'Lorem ipsum dolor sit amet.';
$authorId = 1;
$sql = "INSERT INTO post(title, description, created_at, autho
VALUES( :title,:description , NOW(), :author_id)"; //:title, :

$stmt = $dbh->prepare( $sql );
//Liaison des données avec des paramètres nommés
$result= $stmt->execute([
  ':title'        =>  $title,
  ':description'  =>  $content,
  ':author_id'        =>  $authorId,
]);
```

COUCHE D'ACCÈS AUX DONNÉES

REQUÊTE UPDATE AVEC UN TABLEAU DE VALEURS(MARQUEURS)

```
$sql ="UPDATE post SET title=?, description=? WHERE id = ?";


$id=1;
$stmt = $dbh->prepare( $sql );
        $stmt->execute([  $title, $content,$id ]);
```

COUCHE D'ACCÈS AUX DONNÉES

REQUÊTE SELECT AVEC UN MARQUEUR DE POSITIONNEMENT

```
$n=20;
$sql = "SELECT * FROM post LIMIT :n ";
$stmt = $dbh->prepare($sql);
$stmt->bindValue(':n', $n, PDO::PARAM_INT);

$stmt->execute();
echo "
```

```
";
print_r($stmt->fetchAll(PDO::FETCH_OBJ));
```

LIMITES DE PDO

REQUÊTE SPÉFICIQUE À CHAQUE MOTEUR DE BASE DE DONNÉES

Avec PDO, il est nécessaire d'écrire soi-moi les requêtes SQL pour interroger la base de données. Or les requêtes sont spéficiques à chaque BD bien que le SQL soit un langage standard et normalisé.
Chaque moteur de BD(ou chaque version) propose ses propres fonctionnalités pour interroger la BD.

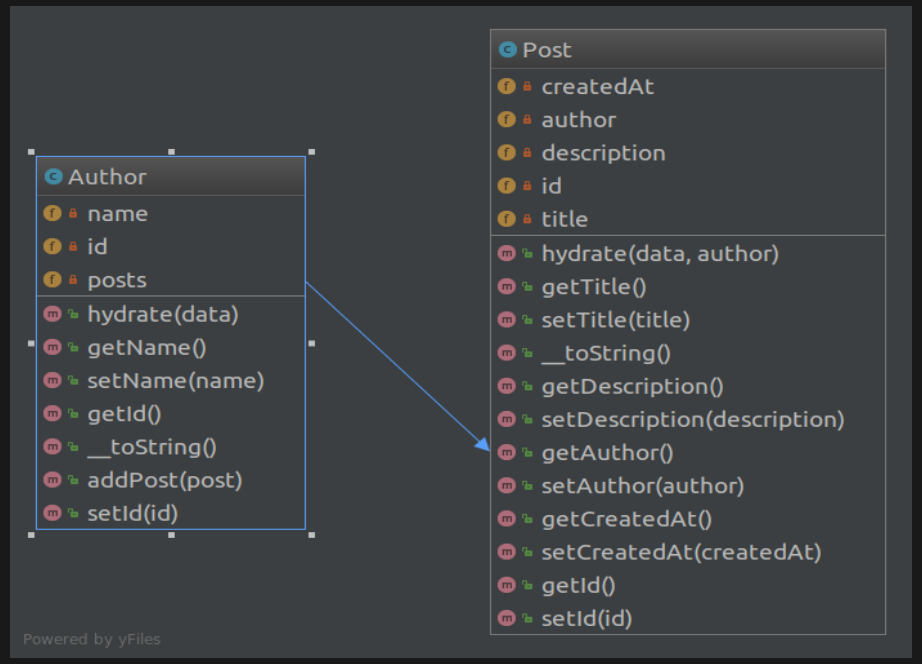
MAPPING OBJET RELATIONNEL(ORM) ET DATA ACCESS LAYER

Un ORM(Object-Relational Mapping) est une interface logicielle permettant d'établir la correspondance entre le modèle objet(classes, interfaces) et le modèle relationnel. Ainsi une table, est perçue comme un objet de PHP et un enregistrement comme attribut de l'objet. L'avantage d'un ORM c'est de s'abstraire complètement de la technologie du moteur de la BD et de ne travailler qu'avec les objets.

EXEMPLE MAPPING OBJET RELATION

Classes PHP

Tables:BD



EXEMPLE MAPPING OBJET(1). CLASSE AUTHOR

```php
class Author {

private $id;
private $name;
/**
        * @var array

        */
private $posts;

/**
 *  Injecte données dans des setters de la classe
 * @param array $data données à injecter
 */
```

Un auteur a plusieurs posts.'Plusieurs' implique un tableau.ici on a un tableau d'ojets Post

EXEMPLE MAPPING OBJET(2). CLASSE POST

```php
class Post {

private $id;
private $title;

private $description;
private $createdAt;
/**
 * @var Author
 */
private $author;
```

DATA ACCESS LAYER(DAL)

Une DAL est une couche d'abastraction, plus elevée que PDO, simplifiant l'accès aux données.

- La couche DAL peut exécuter la même requête SQL pour differents moteurs de BD(pas besoin de requêtes spéficiques à chaque SGBD)
- Basiquement la DAL expose des méthodes génériques pour réaliser opérations CRUD
- Facilite le changement du moteur de BD.

EXEMPLE DAL

```php
class DB {

protected static $instance = null;


/**
* @return \PDO
*/
public static function instance():\PDO {

$config = parse_ini_file(dirname(__DIR__, 2) . "/config/parame
```

TRAVAUX PRATIQUES MAPPING ET DAL

ORM: DOCTRINE

doctrine est un ORM de mapping objet/relationnel transformant la représentation physique des données en une représentation objet et inversement. doctrine apporte une couche d'abstraction qui permet au développeur de se focaliser sur la partie métier de l'application.Il propose des méthodes génériques pour interroger une BD sans écrire une requête SQL. Pour interroger une BDdoctrine utilise un langage intermediaire appelé **DQL(doctrine query language)** qui est traduit en SQL spécifique à la BD sous-jancente: MySQL, MS SQL Server... Doctrine protège sur les données venant directement de l'exterieur vers la BD,cela permet d'éviter une attaque par injection SQL.
Une classe mappée par doctrine s'appelle **entity ou entité**

Supposons que nous souhaitons créer une table en BD post(id , title, description, createdAt)

De table post il en découle l'entité Post contenant les différentes informations dans des propriétés:

```php
/**
 * @ORM\Entity()
 * @ORM\Table(name="post")
 */

class Post {

/**
 * @ORM\Id
 * @ORM\Column(type="integer")
 * @ORM\GeneratedValue
 */
 protected $id;

 /**
```

ANNOTATIONS SUR LA CLASSE

```
1 /**
2  * @ORM\Entity()
3  * @ORM\Table(name="post")
4  */
5 class Post {
6 //...
7 }
```

Les **annotations** sont des blocs de commentaires qui permettent d'ajouter des métadonnées à une classe, à des attributs ou à des méthodes.

Ligne 2: la classe Post est maintenant une entité doctrine. Cette classe est configurée pour faire le lien avec la table post.

Ligne 3: cette annotation est optionnelle. elle intervient que si on veut donner un nom à la table(en BD). Sinon par défaut(absence d'annoation) la table BD prendra le nom del'entité donc Post.

ANNOTATIONS SUR ATTRIBUTS(1): @ORM\COLUMN

Cette annotation configure les colonnes de la table associée à l'entité.

```php
/**
 * @ORM\Id
 * @ORM\Column(type="integer")
 * @ORM\GeneratedValue
 */
protected $id;
```

Cette annotation possède plusieurs proprietés: **type**,**name**,**length**,**unique**,**precision**,**scale**

- type: cette proprieté définit le type de données de la colonne. exemples: integer, string, text, datetime, date,json, boolean, decimal,json...
- name: définit le nom de la colonne.Dans l'exemple précédent, cette proprieté étant absente doctrine attribuera le nom de la proprieté(sans $) au nom de la colonne
- length: longueur maximale pour une colonne de type string. valable que pour le type string.

ANNOTATIONS SUR ATTRIBUTS(2): @ORM\COLUMN

```php
//class Post

/**
 * @Column(type="decimal", precision=6, scale=2, name="price
 */
protected $price;
```

- precision et scale: valables uniquement pour les types 'decimal' et 'float'.precision indique le nombre total de chiffres(avant et après la virgule) et scale le nombre de chiffres après la virgule. ex : 1000,99 (soient 6 chiffres)
- nullable: indique si la colonne est obligatoire. Par défaut elle ne l'est pas.

ANNOTATIONS SUR ATTRIBUTS(3): CLÉ PRIMAIRE

L'annotation @ORM\Id correspond à la clé primaire. Elle se place sur une proprieté voire sur plusieurs proprietés(clés composées)

```php
/**
 * @ORM\Id
 * @ORM\Column(type="integer")
 * @ORM\GeneratedValue
 */
protected $id;
```

@ORM\GeneratedValue utilisée conjointement avec @ORM\Id. Elle spécifie la stratégie de génération de la clé par la BD(auto increment pour MySQL, SEQUENCE pour PostgresSQL ou SQL Server...)

GÉNÉRATION DE LA BASE

Pour générer les tables associées aux entités:

```
vendor/bin/doctrine orm:schema-tool:update --dump-sql --force
```

```
create table post
(
    id int auto_increment
    primary key,

    title varchar(255) not null,
    description longtext not null,
    createdAt datetime null
)
engine=InnoDB collate=utf8_unicode_ci;
```

doctrine indique que la table post a été créée dans la BD

La commande suivante permet de vérifier que le mapping est correct et que les entités sont synchronisées avec la BD

```
vendor/bin/doctrine orm:validate-schema
```

GÉNÉRATION DES REQUÊTES

ENTITYMANAGER

L'entityManager en sigle EM est le composant qui fait le lien entre une entité et sa table associée c'est lui qui orchestre le Mapping. L'EM s'appuie sur le design pattern Data Mapper(derniers cours). Toutes les requêtes(CRUD) passent par l'EM.

Lister tous les articles

```
//retourne tableau d'objets Post
$posts = $entityManager->getRepository(Post::class)->findAll()
```

Lister un article ayant l'id 9

```
$id = 9;//clé primaire
//retourne objet Post
$post = $entityManager->getRepository(Post::class)->find($id);
```

Lister les articles par ordre antichronologique(du plus récent au plus vieux) ayant pour tittre "The title number #57"

ASSOCIATIONS

Les relations entre entités sont materialisées par les annotation suivantes:

- @ORM\OneToOne: 1 entité A est associé à une 1 entité B (1-1)

- @ORM\OneToMany: 1 entité A est associé à plusieurs entités B (1-*)*

- @ORM\ManyToMany: plusieurs entités A sont associées à plusieurs entités B (-*)*

ASSOCIATION ONETOMANY

Exemple : Un auteur écrit plusieurs articles :

```
class Author {
    /**
     * @ORM\OneToMany(targetEntity=Post::class, mappedBy="auth
     */
    private $posts;
 //...getters & setters
}
```

Côté Post: Plusieurs articles sont écrits par 1 auteur :

```
class Post {

    /**
     * @ORM\ManyToOne(targetEntity=Author::class, inversedBy="
     */
    private $author;
//...getters & setters
}
```