L'accès aux données avec PDO

Section 1 - Présentation

Nous avons besoin d'utiliser des données stockées dans une base de données.
PHP peut utiliser un grand nombre de SGBD (système de bases de données) et s'utilise avec MySQL mais aussi Oracle, SQLServer etc.... Nous utiliserons le SGBD Mysql dans ce cours.

Pour se connecter à MySQL via PHP on utilise une API (Application Programming Interface, ou interface de programmation d'application) qui définit les classes, méthodes, fonctions et variables dont votre application va faire usage pour exécuter différentes tâches.

Il y a trois API principales pour se connecter à MySQL :

- L'extension mysql
- L'extension mysqli
- **PHP Data Objects (PDO)**

Les 2 premières sont obsolètes et ne seront plus maintenues, nous utiliserons donc PDO.
PDO est une extension PHP pour formaliser les connexions de base de données de PHP en créant une interface uniforme. Cela permet aux développeurs de créer du code qui est portable sur de nombreuses bases de données et plates-formes.

Section 2 - Utilisation de PDO

A - Connexion à la base de données

Chaque interaction avec une base de données commence par une connexion.
Nous considérons une base de données sur le serveur en local dont le nom est « biblio », nous écrirons le script de connexion ci-dessous dans un fichier "connexionPdo.php" qui sera utilisé par inclusion dans les autres fichiers :

```php
<?php
$hostnom = 'host=localhost'; // nom ou IP du serveur
$usernom = 'root'; // nom de l'utilisateur
$password = '';// mot de passe de l'utilisateur
$bdd = 'biblio'; // nom de la base de données

try {
    // permet de se connecter à la base de données
    $monPdo = new PDO("mysql:$hostnom;dbname=$bdd;charset=utf8", $usernom, $password);
    // permet d'activer la gestion des erreurs en les affichant à l'écran
    $monPdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
} catch (PDOException $e) {
    echo $e->getMessage();
    $monPdo = null; // ferme la connection à la base de données
}
?>
```

le try…catch permet d'intercepter une erreur. Si la connexion n'aboutit pas (dans le try), un message d'erreur est affiché via le catch de l'exception

affichage des erreurs (importante lors du développement !)

Pour fermer la connexion, l'objet doit être détruit. Cela se fait normalement à la fin d'un script où PHP fermera automatiquement la connexion. Cependant, la connexion peut être fermée explicitement

B - Gestion des erreurs

Comme vous avez pu le voir sur l'extrait de code, nous utilisons les try…catch pour intercepter les erreurs.
Pour cela il faut un attribut de gestion des erreurs (`PDO::ATTR_ERRMODE`). Il existe trois options pour gérer les erreurs:

`PDO::ERRMODE_SILENT`

C'est le mode par défaut. PDO définit simplement le code d'erreur à inspecter grâce aux méthodes **PDO::errorCode()** et **PDO::errorInfo()** sur les objets représentant les requêtes, mais aussi ceux représentant les bases de données ;

`PDO::ERRMODE_WARNING`

En plus de définir le code d'erreur, PDO émettra un message **E_WARNING** traditionnel. Cette configuration est utile lors des tests et du débogage, si vous voulez voir le problème sans interrompre l'application.

`PDO::ERRMODE_EXCEPTION`

En plus de définir le code d'erreur, PDO lancera une exception **PDOException** et récupérera les propriétés de l'erreur pour obtenir des informations complémentaires. Cette configuration est très utile lors du débogage, car elle va vous permettre de catcher l'erreur au point critique de votre code pour vous montrer rapidement le problème rencontré (les transactions sont automatiquement annulées si l'exception fait que votre script se termine cf ci-après). Biensur PHP n'execute pas la suite du script qui a provoqué l'erreur.

C - Les requêtes Action

Ce sont les requêtes qui modifient la base données: insert, update, delete. Ces trois types de requêtes se traitent de la même manière. Voici l'exemple d'un traitement d'une requête de type « `INSERT` » :

```
try{
include('connexionPdo.php');
// on prépare la requête (mais elle n'est pas encore exécutée)
$req = $monPdo->prepare("insert INTO genre (libelle) VALUES ('Espionnage')" );

// on execute la requête et récupérons le résultat (un entier qui décrit
// le nombre de lignes affectées ou zéro si l'insertion n'a pas fonctionné)
$nbr = $req->execute();

echo "nombre d'insertions :".$nbr."<br />" ;
//affichage de l'identifiant créé automatiquement (cas d'un autoincrément)
$Id = $monPdo->lastInsertId();
echo "le numéro attribué est : ".$Id;
//*** fermer la connexion *** /
$monPdo=null ;
}
catch ( PDOException $e )
{
echo $e->getMessage () ;
$monPdo=null ;
```

Permet de récupérer l'id attribué par le SGBDR dans le cas d'un auto_increment

Pour des requêtes « `UPDATE` » il suffit de changer le texte de la requête comme ceci par exemple :

```
$req = $monPdo->prepare("update genre SET libelle='Horreur' WHERE libelle ='Terreur' ");
```

Pour des requêtes « `DELETE` » il suffit de changer le texte de la requête comme ceci par exemple :

```
$req = $monPdo->prepare("delete from genre where libelle='Espionnage'");
```

D - Les requêtes Sélection

Contrairement aux requêtes action les requêtes sélection renvoient un ensemble de résultats, sous la forme d'un objet `PDOStatement`. Il faut parcourir ce résultat en utilisant un `fetch` pour lire une ligne ou un `fetchAll` pour récupérer l'ensemble des lignes en une seule fois (cf ci-après).
Prenons l'exemple d'une requête permettant d'avoir la liste de tous les auteurs :

```
$req = $monPdo->prepare("SELECT * FROM auteur"); // on prépare la requête
$req->execute();// on execute la requête
```

Voici un extrait des résultats :

num	nom	prenom	nationalite
1	Dostoeïvski	Fédor	Russe
2	Semprun	Jorge	Espagnol
3	Tolstoï	Leon	Russe
4	Steinbeck	John	Américain
5	Ferro	Marc	Français

D.1 - Récupération des données et mode fetchAll (résultat de toute la requête)

Le principe est de récupérer tous les résultats en mémoire (attention dans le cas de grosses requêtes) et de parcourir ensuite chaque ligne, et , pour chaque ligne parcourir l'ensemble des valeurs des différentes colonnes.

Voici un exemple :

```php
$req->setFetchMode(PDO::FETCH_ASSOC);
$lesLignes = $req->fetchAll();
foreach ($lesLignes as $laLigne) {
    echo "<tr>";
    echo "  <td>" . $laLigne['num'] . "</td>
            <td>" . $laLigne['nom'] . "</td>
            <td>" . $laLigne['prenom'] . "</td>
            <td>" . $laLigne['nationalite'] . "</td>";
    echo "</tr>";
}
```

Avec **FETCH_ASSOC** les clés sont les noms de champs

```
Array
(
    [0] => Array
        (
            [num] => 1
            [nom] => Dostoeïvski
            [prenom] => Fédor
            [nationalite] => Russe
        )

    [1] => Array
        (
            [num] => 2
            [nom] => Semprun
            [prenom] => Jorge
            [nationalite] => Espagnol
        )
```

OU

```php
$req->setFetchMode(PDO::FETCH_NUM);
$lesLignes = $req->fetchAll();
foreach ($lesLignes as $laLigne) {
    echo "<tr>";
    echo "  <td>" . $laLigne[0] . "</td>
    <td>" . $laLigne[1] . "</td>
    <td>" . $laLigne[2] . "</td>
    <td>" . $laLigne[3] . "</td>";
    echo "</tr>";
}
```

Avec **FETCH_NUM** les clés sont des indices numériques

```
Array
(
    [0] => Array
        (
            [0] => 1
            [1] => Dostoeïvski
            [2] => Fédor
            [3] => Russe
        )

    [1] => Array
        (
            [0] => 2
            [1] => Semprun
            [2] => Jorge
            [3] => Espagnol
        )
```

OU

```php
$req->setFetchMode(PDO::FETCH_OBJ);
$lesLignes = $req->fetchAll();
foreach ($lesLignes as $laligne) {
    echo "<tr>";
    echo "  <td>" . $laLigne->num . "</td>
            <td>" . $laLigne->nom . "</td>
            <td>" . $laLigne->prenom . "</td>
            <td>" . $laLigne->nationalite . "</td>";
    echo "</tr>";
}
```

Avec **FETCH_OBJ** les données sont transformées en objet (objet anonyme stdClass)

```
array(33) {
  [0]=>
  object(stdClass)#3 (4) {
    ["num"]=>
    string(1) "1"
    ["nom"]=>
    string(12) "Dostoeïvski"
    ["prenom"]=>
    string(6) "Fédor"
    ["nationalite"]=>
    string(5) "Russe"
  }
  [1]=>
  object(stdClass)#4 (4) {
```

OU

```php
$req->setFetchMode(PDO::FETCH_CLASS, 'Auteur');
$lesLignes = $req->fetchAll();
foreach ($lesLignes as $laLigne) {
    echo "<tr>";
    echo "  <td>" . $laLigne->getNum() . "</td>
            <td>" . $laLigne->getNom() . "</td>
            <td>" . $laLigne->getPrenom() . "</td>
            <td>" . $laLigne->getNationalite() . "</td>";
    echo "</tr>";
}
```

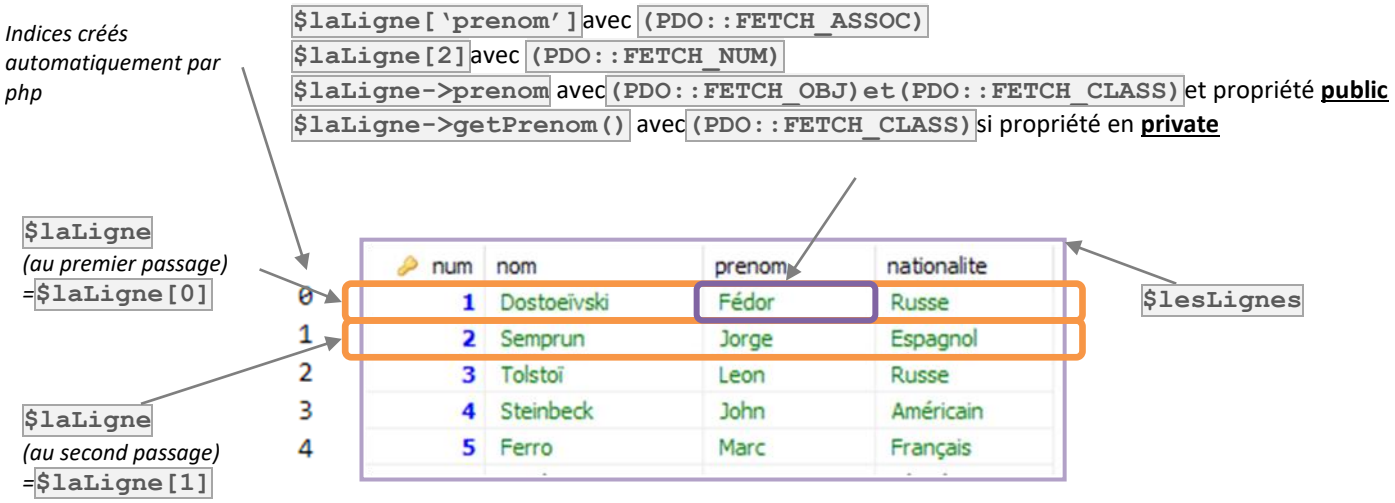
Avec **FETCH_CLASS** les données sont mappées avec les propriétés de la classe précisée en second paramètre (si les données sont private, alors on doit passer par les getters).

```php
class Auteur
{
    private $num; // numéro de l'auteur
    private $nom; // nom de l'auteur
    private $prenom; // nom de l'auteur
    private $nationalite; // nom de l'auteur

    // getters pour les données privées
    public function getNum(){
        return $this->num;
    }
    public function getNom(){
        return ucwords($this->nom);
    }
    public function getPrenom(){
        return $this->prenom;
    }
    public function getNationalite(){
        return ucwords($this->nationalite);
    }
}
```

```
 [0]=>
 object(Auteur)#36 (4) {
   ["num":"Auteur":private]=>
   string(1) "1"
   ["nom":"Auteur":private]=>
   string(12) "Dostoeïvski"
   ["prenom":"Auteur":private]=>
   string(6) "Fédor"
   ["nationalite":"Auteur":private]=>
   string(5) "Russe"
 }
```

Schématisation en mémoire :



D.2 - Récupération des données et mode fetch (résultat d'une ligne à la fois)

Le principe est de récupérer une seule ligne à la fois en mémoire.
Le fetch fonctionne avec les mêmes constantes vues précédemment et passées en paramètre (`FETCH_ASSOC` et `FETCH_NUM` ,`FETCH_OBJ` et `FETCH_CLASS`).

D.2.1 - Méthode longue

Exemple avec `FETCH_OBJ` mais le principe est le même avec `FETCH_ASSOC` `FETCH_NUM` et `FETCH_CLASS`):

```
echo "<table>";
$laLigne = $req->fetch();                    // on lit la première ligne
while ($laLigne != null) { // tant que une ligne est lue ($laLigne est différent de null)
    echo "<tr>";
    echo "   <td>" . $laLigne->num . "</td>
           <td>" . $laLigne->nom . "</td>
           <td>" . $laLigne->prenom . "</td>
           <td>" . $laLigne->nationalite . "</td>";
    echo "</tr>";
    $laLigne = $req->fetch();                // on lit la ligne suivante
}
echo "</table>";
```

Les valeurs ont été tranformées en propriété de l'objet et on y accède avec le signe '->'

D.2.2 - Méthode courte

Exemple avec `FETCH_OBJ` mais le principe est le même avec `FETCH_ASSOC` `FETCH_NUM` et `FETCH_CLASS`):

```
echo "<table>";
while ($laLigne = $req->fetch()){                    { // tant qu'on arrive à lire une ligne
    echo "<tr>";
    echo "   <td>" . $laLigne->num . "</td>
           <td>" . $laLigne->nom . "</td>
           <td>" . $laLigne->prenom . "</td>
           <td>" . $laLigne->nationalite . "</td>";
    echo "</tr>";
}
```

Dans ce cas on économise 2 lignes de code car on lit directement l'occurrence de la requête dans le while !

D.3 - Cas particulier : POO et Fetch Class Avec FETCH_PROPS_LATE

Les noms des champs sont mappés aux propriétés de la classe appelée. Le ou les objets vont être instanciés directement grâce à la classe `Auteur`. Pour chaque élément instancié PHP va donc faire appel au `constructeur` de la classe (ce qui peut pénaliser en terme de ressource temps sur de grosses requêtes).

Prenons l'exemple d'u parcours d'objets de la classe Auteur :

```php
include "connexionPDO.php";//connexion
include "auteur.class.php";// fichier de la classe auteur
$req = $monPdo->prepare("SELECT * FROM auteur"); // on prépare la requête
$req->setFetchMode(PDO::FETCH_CLASS, 'Auteur');
$req->execute();// on execute la requête
echo "<table>";
$lesLignes = $req->fetchAll();
foreach ($lesLignes as $laLigne) {
    echo "<tr>";
    echo "   <td>" . $laLigne->num . "</td>
             <td>" . $laLigne->nom . "</td>
             <td>" . $laLigne->prenom . "</td>
             <td>" . $laLigne->nationalite . "</td>";
    echo "</tr>";
}
echo "</table>";
```

Ne pas oublier l'include ou le use

```php
class Auteur
{
    private $num; // numéro de l'auteur
    private $nom; // nom de l'auteur
    private $prenom; // nom de l'auteur
    private $nationalite; // nom de l'auteur

    // getters pour les données privées
    public function getNum(){
        return $this->num;
    }
    public function getNom(){
        return ucwords($this->nom);
    }
    public function getPrenom(){
        return $this->prenom;
    }
    public function getNationalite(){
        return ucwords($this->nationalite);
    }
}
```

Fatal error: Cannot access private property genre::$num

Ce code génère une **Fatal Error** car nous tentons de faire appel aux propriétés alors qu'elles sont en private, il faut donc faire appel aux méthodes getters comme le suppose la class, comme ceci :

```php
<td>" . $laLigne->getNum() . "</td>
<td>" . $laLigne->getNom() . "</td>
<td>" . $laLigne->getPrenom() . "</td>
<td>" . $laLigne->getNationalite() . "</td>";
```

1	Dostoeïvski	Fédor	Russe
2	Semprun	Jorge	Espagnol
3	Tolstoï	Leon	Russe
4	Steinbeck	John	Américain

Remarquez également qu'en passant par la méthode il met bien grâce à la fonction « `ucwords` »utilisée dans les getters, la première lettre des nom, prénom et nationalité en majuscule

`FETCH_CLASS` a récupéré les résultats directement dans la classe Auteur et a construit les objets ce qui permet de manipuler directement les résultats. Le constructeur est appelé après la construction des objets !

Attention :

Si vous voulez imposer le passage par le constructeur avant l'affectation des valeurs aux propriétés de l'objet, vous devez alors utiliser `FETCH_CLASS` avec `FETCH_PROPS_LATE` ensemble comme ci-dessous :

```php
$req->setFetchMode(PDO::FETCH_CLASS | PDO::FETCH_PROPS_LATE, 'Auteur');
```

Section 3 - Les requêtes préparées

Une requête préparée est une déclaration" précompilée" qui accepte des paramètres nommés ou paramètres non définis lors de la pré-compilation. Les requêtes préparées sont non seulement **plus sures** car elles évitent les injections SQL mais aussi plus portables, plus souples , et bien **plus rapides** à exécuter parce qu'elles sont déjà traduites.

Il existe deux types de marqueurs de paramètres.

- paramètre nommé – « `: nomDuParametre` »
- paramètre de type marqueur : « `?` »

le paramètre sera remplacé à l'exécution par sa valeur et vous devez choisir l'une ou l'autre des méthodes, elles ne peuvent pas être mélangées.

A - Execution d'une requête avec paramètre nommé

```php
// on prépare la requête paramètrée
$req = $monPdo->prepare("SELECT * FROM genre where libelle like :paramNom and num > :paramId ");
$valeurNom = "%er%"; // on peut bien sur récupérer cette valeur avec des $_POST ou $_GET
$valeurId = 3;
// on alimente les paramètres
// attention le premier argument est entre côte et le second est une variable et non une valeur
$req->bindParam(':paramNom', $valeurNom);
$req->bindParam(':paramId', $valeurId);
//on execute la requête (mais on ne récupére pas le résultat encore !)
$req->execute();
```

B - Execution d'une requête avec paramètre de type marqueur

```php
// on prépare la requête paramètrée
$req = $monPdo->prepare("SELECT * FROM genre where libelle like ? and num > ? ");
$valeurNom = "%er%"; // on peut bien sur récupérer cette valeur avec des $_POST ou $_GET
$valeurId = 3;
// on alimente les paramètres avec un array même temps que l'execution
//on execute la requête (mais on ne récupére pas le résultat encore !)
$req->execute(array($valeurNom, $valeurId));
```

C - Exemple qui montre la réutilisabilité de la requête préparée

```php
$req = $dbh->prepare("INSERT INTO genre (num, libelle) VALUES (:num, :libelle)");
$req->bindParam(':num', $num);
$req->bindParam(':libelle', $libelle);

// insertion d'une ligne
$num = '10'; // on voit ici que la valeur des paramètres est alimentée
//après le bindParam
$libelle = 'Comédie';
$req->execute();

// insertion d'une autre ligne avec des valeurs différentes
$name = '11';
$value = 'Horreur';
$req->execute(); // on réexécute la requête sans la "re préparer"
// gain enorme de performance !
```

Section 4 - Les transactions

Une transaction est un groupe d'instruction qui doivent être effectuées **ensemble ou pas du tout**.

L'exemple le plus connu est celui du retrait d'un compte pour créditer un autre compte. Si le retrait ne s'effectue pas correctement, il ne doit pas y avoir de crédit de l'autre compte.

Il s'agit donc de pouvoir effectuer les mises à jour quand toutes les opérations sont effectuées ou de les annuler **globalement**.

Les transactions offrent 4 fonctionnalités majeures : **Atomicité**, **Consistance**, **Isolation** et **Durabilité** (**ACID**). Les transactions sont typiquement implémentées pour appliquer toutes vos modifications en une seule fois ; ceci a le bel effet d'éprouver drastiquement l'efficacité de vos mises à jour.

Les transactions rendent vos scripts plus rapides et potentiellement plus robustes. Exemple on veut créer une nouvelle table et insérer des données. Si la création de table n'aboutit pas ou qu'un ou plsieurs insert plantent, les données ne sont alors plus complètes ou cohérentes suivant le cas, on veut annuler tout. Nous allons utiliser une transaction.

Une transaction de PDO commence avec la méthode `beginTransaction()` . Cette méthode désactive l'auto-validation et les exécutions de la méthode `execute` sur la base de données ne sont pas exécutées tant que la transaction n'est pas validée avec `commit()` .

En cas de problème, un `rollback()` est exécuté qui annule toutes les actions depuis le begin Transaction. Cela permet de garder la base de données dans un état stable avec des données correctes.

```php
include "connexionPDO.php";
try{
// *** commencer la transaction après avoir demandé les exceptions*** /
$monPdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$monPdo->beginTransaction ();
$req = $monPdo->prepare("INSERT INTO genre (num, libelle) VALUES (:num, :libelle)");

$req->bindParam(':num', $num);
$req->bindParam(':libelle', $libelle);

// insertion d'une ligne
    $num = '10';
    $libelle = 'Comédie';
    $req->execute();

// insertion d'une autre ligne avec des valeurs différentes
    $name = '11';
    $value = 'Horreur';
    $req->execute(); // on réexécute la requête sans la "re préparer"
//et encore plein d'autres insertions si l'on veut

// * ** valider la transaction *** /
$monPdo->commit ();
// *** afficher un message  ok* /
echo    'Les données saisies ont été crée
}
catch (PDOException $e)
{
// *** rollback de la transaction en cas d'erreur *** /
$monPdo->rollback();
echo "<br>"."les modifications ne sont pas faites pour cause d'erreur";
// *** écho l'instruction SQL et le message d'erreur *** /
echo    '<br>'.$e->getMessage();
}
```

On démarre la transaction

Les requêtes sont enregistrées mais pas exécutées sur la base de données tant que le commit n'est pas appelé

Ceci indique que l'ensemble des requêtes depuis le `beginTransaction()` seront appliquées d'un coup, si l'une d'entre elles échoue alors la base de données sera remise dans son état initial

Section 5 - Le pattern Singleton

L'objectif d'un singleton est qu'il n'y ait qu' **une seule instance de la base** pour fournir un point d'accès global.
Nous utilisons la méthode `getInstance()` pour atteindre cet objectif. Une nouvelle instance est créée uniquement la première fois et elle est accessible pour tous les accès suivants car si elle existe déjà, l'instance existante est simplement renvoyée.

Voici un exemple de fichier qui utilise la classe **MonPdo** :

```php
try {
    include('connexionPdo.php');
    $monPdo = MonPdo::getInstance();
    $req = $monPdo->prepare("insert INTO genre (libelle) VALUES ('Espionnage')");
// ....

} catch (PDOException $e) {
    echo $e->getMessage();
}
```

On récupère une instance unique de PDO

et voici la classe (dans le fichier connexionPDO) qui génère une seule instance d'elle-même :

```php
class MonPdo
{
    private static $serveur='mysql:host=localhost';
    private static $bdd='dbname=animaux'; private static $user='root'  ; private static $mdp='' ;
    private static $unPdo=null;
    //sert pour l'appel du constructeur qui renseigne $unPdo
    private static $monPdo ;
    /**Constructeur*/
    private function __construct()
    {
    MonPdo::$unPdo = new PDO(MonPdo::$serveur.';'.MonPdo::$bdd, MonPdo::$user, MonPdo::$mdp);
    MonPdo::$unPdo->query("SET CHARACTER SET utf8");
    MonPdo::$unPdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
    }
    /**Desstructeur*/
    public function __destruct(){
    MonPdo::$unPdo = null;
    }
    /**
    *Fonction statique qui crée l'unique instance de la classe
    * Appel : $instance = MonPdo::getInstance();
    * @return l'unique objet de la classe MonPdo
    */
    public  static function getInstance()
    {
        if(MonPdo::$unPdo == null)
        {
            MonPdo::$monPdo= new MonPdo();
        }
        return MonPdo::$unPdo; // dans tous les cas retourne l'instance
    }
}
```