

PHP/PDO : accès en lecture/ écriture

Table des matières

I. Contexte	3
II. Récupérer des données	3
III. Exercice : Appliquez la notion	6
IV. Les requêtes préparées	6
V. Exercice : Appliquez la notion	9
VI. PDOStatement	9
VII. Exercice : Appliquez la notion	15
VIII. Récupérer le résultat	15
IX. Exercice : Appliquez la notion	20
X. La pagination	20
XI. Exercice : Appliquez la notion	22
XII. Essentiel	23
XIII. Auto-évaluation	23
A. Exercice final	23
B. Exercice : Défi	25
Solutions des exercices	26

I. Contexte

Durée : 1 h

Environnement de travail : Local

Pré-requis : Connaissances en SQL et PDO

Contexte

Tout l'intérêt d'une base de données est de pouvoir récupérer efficacement les données qu'elle contient, via des requêtes `SELECT`, pour pouvoir ensuite les manipuler dans notre application. Dans ce cours, nous allons voir les différentes manières de procéder en PHP et les avantages et inconvénients de chaque méthode.

II. Récupérer des données

Objectifs

- Lancer une requête `SELECT` depuis notre application PHP
- Afficher le résultat

Mise en situation

La plupart des sites web doivent gérer des utilisateurs : c'est un cas d'utilisation très fréquent que nous allons utiliser comme exemple dans ce cours. Considérons donc qu'une table de notre base de données permette de stocker les informations de nos utilisateurs.

users	
<u>id</u>	INT(11)
name	VARCHAR(100)
password	VARCHAR(100)
email	VARCHAR(100)

Notre premier but va être d'afficher la liste de tous les utilisateurs dans notre application PHP.

Méthode Lancer une requête `SELECT`

La manière la plus simple de récupérer des données depuis une base de données grâce à PDO est d'utiliser la méthode `query`. Elle prend en paramètres une requête SQL et un mode de récupération et retourne le résultat de la requête sous un format qui dépend du mode choisi.

Pour le moment, nous allons utiliser le mode `PDO::FETCH_ASSOC`, qui signifie que chaque ligne de données sera sous forme de tableau associatif dont les clés ont le même nom que les colonnes récupérées dans le `SELECT`.

```
[id] => 1
[username] => john
[password] => a94a8fe5ccb19ba61c4c0873d391e987982fbbd3
```

Il suffit ensuite de boucler sur les résultats pour manipuler chaque ligne. Pour cela, il est possible d'utiliser la méthode `query` directement dans un `foreach` :

```
1 <?php
2 // Initialisation de l'objet PDO, construction de la requête...
3 foreach ($pdo->query($sqlRequest, PDO::FETCH_ASSOC) as $row) {
4     // Ici, la variable $row est un tableau associatif
5 }
```

Exemple

Pour afficher la liste de tous nos utilisateurs, il suffit d'envoyer une requête de sélection sur la table `User` et de boucler sur le résultat. Par exemple, le code suivant permet d'afficher les noms et e-mails de chaque utilisateur :

```
1 <?php
2 try {
3     $pdo = new PDO('mysql:host=localhost;dbname=php_app', 'root', '');
4     foreach ($pdo->query('SELECT name, email FROM users', PDO::FETCH_ASSOC) as $user) {
5         echo $user['name'].' '.$user['email'].'<br>';
6     }
7 } catch (PDOException $e) {
8     echo 'Impossible de récupérer la liste des utilisateurs';
9 }
```

Remarque

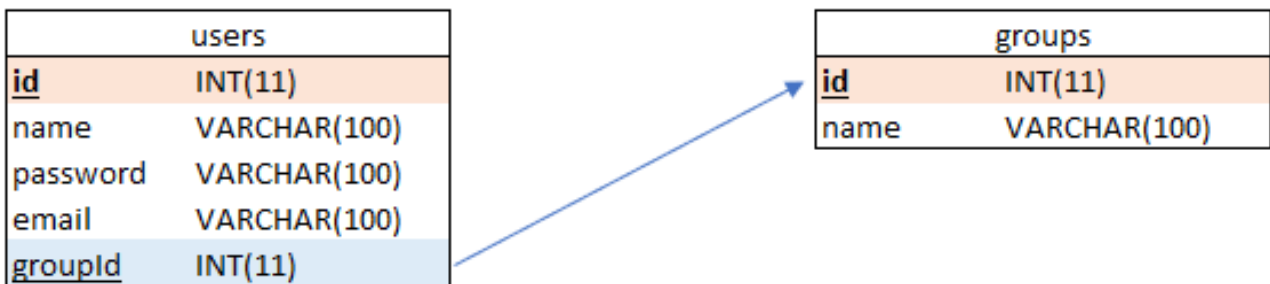
Il est possible d'utiliser la méthode `query()` pour des requêtes qui ne sont pas des requêtes de sélection, mais dans ce cas, le résultat sera vide. Pour les autres requêtes, il est recommandé d'utiliser `exec()`, qui retourne le nombre de lignes affectées.

Fondamental Jointures et alias

Chaque clé du tableau associatif est basée sur le champ présent dans le `SELECT` et non pas sur le nom de la colonne de la table. Même si, dans de nombreux cas, ces deux valeurs sont similaires, cela signifie qu'il est possible de changer les colonnes du tableau grâce à l'utilisation d'**alias** dans le `SELECT`.

```
1 <?php
2 try {
3     $pdo = new PDO('mysql:host=localhost;dbname=php_app', 'root', '');
4     // On utilise un alias, grâce au mot-clé AS
5     foreach ($pdo->query('SELECT name AS nom, email AS adresse FROM users', PDO::FETCH_ASSOC)
6 as $user) {
7         // Ici, on utilise le nom de l'alias et non celui de la colonne
8         echo $user['nom'].' '.$user['adresse'].'<br>';
9     }
10 } catch (PDOException $e) {
11     echo 'Impossible de récupérer la liste des utilisateurs';
12 }
```

Or, dans un tableau associatif, il ne peut pas y avoir deux clés ayant le même nom. Cela signifie que, si dans la requête `SELECT` deux colonnes portent le même nom, alors l'une d'entre elles sera écrasée. Par exemple, admettons que nos utilisateurs puissent appartenir à un groupe et que chaque groupe possède son propre nom :



La jointure entre ces deux tables, issue de la requête `SELECT * FROM users JOIN groups ON groups.id = users.groupId`, devrait comporter 7 colonnes. Or, le tableau associatif généré par PDO n'en contiendra que 5 : les clés `id` et `name` sont écrasées.

```
array(5) {
    ["id"]=>
    string(1) "1"
    ["name"]=>
    string(15) "administrateurs"
    ["password"]=>
    string(40) "109f4b3c50d7b0df729d299bc6f8e9ef9066971f"
    ["email"]=>
    string(12) "john@doe.com"
    ["groupId"]=>
    string(1) "1"
}
```

Pour éviter cette perte de données, il est possible de renommer les champs en double en utilisant des alias. Ainsi, on peut renommer les champs ayant un nom identique grâce à la requête :

```
1 SELECT
2     users.id AS userId,
3     users.name as userName,
4     users.password,
5     users.email, users.groupId,
6     groups.*
7 FROM users
8 JOIN groups ON groups.id = users.groupId
```

On retrouve alors l'intégralité de nos champs :

```
array(7) {
    ["userId"]=>
    string(1) "1"
    ["userName"]=>
    string(4) "john"
    ["password"]=>
    string(40) "109f4b3c50d7b0df729d299bc6f8e9ef9066971f"
    ["email"]=>
    string(12) "john@doe.com"
    ["groupId"]=>
    string(1) "1"
    ["id"]=>
    string(1) "1"
    ["name"]=>
    string(15) "administrateurs"
}
```

Syntaxe À retenir

- Pour lancer une requête `SELECT`, le moyen le plus simple est d'utiliser la méthode `query`.
- Le mode de récupération `PDO::FETCH_ASSOC` permet de récupérer le résultat sous forme de tableau associatif. Chaque clé du tableau correspond à un champ retourné par le `SELECT`, qu'il est possible de renommer en utilisant des alias, ce qui est pratique en cas de conflit de noms de colonnes.

Complément

La méthode `query`¹

III. Exercice : Appliquez la notion

Question

[solution n°1 p.27]

Un site de test de jeux vidéo aimerait refaire son site en utilisant PDO. Les données sont stockées dans une base de données qu'il est possible d'importer via le script suivant :

```
1 CREATE TABLE games(
2     id INTEGER(11) PRIMARY KEY AUTO_INCREMENT,
3     name VARCHAR(500),
4     description VARCHAR(500)
5 );
6 INSERT INTO games(name, description) VALUES
7 ('Super Moria World', 'Parcourez le monde pour jeter une tortue dans la lave !'),
8 ('The legend of Zebda', 'Une épopée musicale où vous devez vaincre les forces du mal.');
```

En utilisant vos nouvelles connaissances, créez une page permettant d'afficher la liste des jeux et leur description. En cas d'erreur de connexion, affichez simplement un message d'erreur.

IV. Les requêtes préparées

¹ <https://www.php.net/manual/fr/pdo.query.php>

Objectifs

- Préparer une requête
- Comprendre la notion de marqueurs

Mise en situation

La méthode `query` est très utile pour récupérer le résultat de requêtes simples. Cependant, elle montre rapidement ses limites dans des cas plus complexes. Dans cette partie, nous allons voir quelles sont ses limites et comment y remédier.

Les limites de query

Améliorons un peu notre liste d'utilisateurs en ajoutant un champ de recherche sur notre application : un utilisateur pourra ainsi en rechercher un autre par son nom. Nous allons donc devoir améliorer notre requête `SELECT` pour lui ajouter une clause `WHERE` permettant de filtrer les résultats.

Pour cela, nous allons utiliser l'opérateur `LIKE` et injecter la valeur saisie par l'utilisateur, suivie d'un symbole `%`. Ainsi, en recherchant simplement « a », le résultat sera la liste de tous les utilisateurs dont le nom commence par la lettre « a ».

En utilisant la méthode `query`, cela donne le code suivant :

```
1 <?php
2 $pdo->query('SELECT * FROM users WHERE name LIKE \''.$_GET['search'].'%', PDO::FETCH_ASSOC);
```

Cette requête fonctionne pour les recherches simples, mais si l'utilisateur recherche un terme comportant un apostrophe, elle ne sera plus valide et soulèvera une erreur. Pire, un utilisateur malintentionné pourrait utiliser cette faiblesse pour faire de l'injection SQL et manipuler notre base de données.

Nous allons donc devoir gérer nous-mêmes l'échappement des caractères. Le problème est que sa syntaxe dépend du SGBD : certains permettent de doubler l'apostrophe, mais d'autres acceptent également l'antislash, ce qui en fait un processus plus complexe à mettre en place, qu'il faudra répéter pour chaque requête.

Heureusement, PDO peut gérer ça, pour nous, grâce aux requêtes préparées.

Rappel

`$_GET` est un tableau de données associatif et super globale qui va permettre de faire passer des informations de page en page grâce à l'url.

Les injections SQL

Une injection SQL (« Structured Query Language », langage de requête structurée) représente un type de cyberattaque. Le pirate effectuant cette attaque va modifier une requête SQL en cours en injectant un « morceau » de requête, pour manipuler une base de données et accéder à des informations importantes.

Il s'agit d'un des genres d'attaques les plus populaires et menaçants, car il peut être utilisé pour nuire à n'importe quelle application ou site Web qui dispose d'une base de données SQL. L'injection SQL¹ est uniquement réalisable lorsqu'une requête est générée à partir de données fournies par un utilisateur.

Ces données sont directement exploitées pour construire toute une requête ou seulement une partie. Il peut s'agir d'un insert, un update, une jointure, de conditions de filtres ou de regroupement.

Grâce à une utilisation adaptée de l'objet PDO pour préparer les requêtes MySQL en PHP², il est impossible pour une personne mal intentionnée de réaliser des injections SQL.

1 <https://analyse-innovation-solution.fr/publication/fr/hacking/injection-sql-sqli-dorks>

2 <https://analyse-innovation-solution.fr/publications/sql>

Exemple

Ci-dessous se trouve une requête nous permettant de sélectionner et d'afficher toutes les données de la table user où le champs username prendra comme valeur le contenu de la variable \$username et pass, celui de \$password.

```
1 4 $sql = "SELECT * FROM `users` WHERE `username`=' $username' AND `pass`=$password";
2 5 $requete = $db->query($sql);
3 6 $user = $requete->fetchAll();
4 7
5 8 var_dump($user);
```

En remplissant la variable \$username de la manière suivante, nous exploitons une faille.

```
1 $username = "admin'; --";
2 $password = "";
```

En effet, en procédant de cette façon, '--' permettant de créer un commentaire en langage SQL, tout ce qui se trouve à la suite de celui-ci devient commenté. Nous avons donc accès aux informations concernant l'admin, dont son mot de passe.

```
1 $sql = "SELECT * FROM `users` WHERE `username`='admin'; --' AND `pass`=$password";
```

Méthode Requetes préparées

Une requête préparée est une requête possédant des paramètres, dans laquelle nous allons injecter des valeurs. Ainsi, plutôt que de former la requête nous-mêmes en utilisant la concaténation de chaînes de caractères, nous allons placer des marqueurs que PDO remplacera par les valeurs au moment de son exécution.

Pour utiliser une requête préparée, il y a deux étapes à respecter : la première est de préparer la requête, c'est-à-dire déclarer une requête possédant des marqueurs ; il faut ensuite exécuter cette requête, en renseignant la valeur des marqueurs.

On peut voir une requête préparée un peu comme une « requête-fonction » : on doit d'abord la déclarer en indiquant les paramètres attendus en entrée, puis l'appeler en renseignant ses paramètres.

L'avantage d'utiliser des requêtes préparées est qu'au moment où PDO va injecter les valeurs dans nos marqueurs, il va échapper automatiquement tous les caractères qui ont besoin de l'être, en utilisant la méthode d'échappement correspondant au driver utilisé.

Syntaxe Les marqueurs

Il existe deux types de marqueurs que l'on peut utiliser dans une préparation de requête : les marqueurs nommés et les marqueurs interrogatifs.

- Les marqueurs nommés permettent, d'attribuer un nom à chaque paramètre. Ils se déclarent dans la requête grâce à la syntaxe :nomMarqueur. `SELECT * FROM users WHERE name LIKE :search`
- Les marqueurs interrogatifs, quant à eux, vont se baser sur la position du paramètre plutôt qu'un nom. Ils se déclarent dans la requête grâce à un point d'interrogation. `SELECT * FROM users WHERE name LIKE ?`

Attention

Chaque marqueur représente une valeur à part entière. Dans notre condition, nous souhaitons rajouter un % à la fin de notre paramètre, mais ce caractère fera partie de la valeur de notre marqueur. Ainsi, il n'est pas possible d'écrire : `SELECT * FROM users WHERE name LIKE :search%`.

C'est la valeur du paramètre :search qui contiendra le symbole %.

Il n'est pas non plus possible d'utiliser les marqueurs pour remplacer des noms de colonne ou des opérateurs : seulement des valeurs.

Syntaxe À retenir

- Une requête préparée est une requête possédant des paramètres sous la forme de marqueurs.
- Il existe deux types de marqueurs : les marqueurs nommés, qui attribuent un nom à chaque paramètre avec la syntaxe `:nom` et les marqueurs interrogatifs, qui remplacent tous les paramètres par des points d'interrogation.
- L'intérêt des requêtes préparées repose sur le fait qu'au moment de donner une valeur aux paramètres, PDO va automatiquement formater les valeurs selon leur type, incluant l'échappement des caractères qui en ont besoin.

Complément

Les requêtes préparées¹

V. Exercice : Appliquez la notion

Question 1

[solution n°2 p.27]

Reprenons la base de données de notre site de tests de jeux vidéos :

```
1 CREATE TABLE games(  
2     id INTEGER(11) PRIMARY KEY AUTO_INCREMENT,  
3     name VARCHAR(500),  
4     description VARCHAR(500)  
5 );  
6 INSERT INTO games(name, description) VALUES  
7 ('Super Moria World', 'Parcourez le monde pour jeter une tortue dans la lave !'),  
8 ('The legend of Zebda', 'Une épopée musicale où vous devez vaincre les forces du mal.');
```

Les utilisateurs souhaiteraient pouvoir rechercher un jeu en saisissant le début de son nom. Rédigez la requête préparée permettant de faire cette recherche en utilisant des marqueurs nommés.

Indice :

Le fait de vouloir saisir le début du nom du jeu donne une indication sur l'opérateur à utiliser, mais pas sur la valeur du marqueur.

Question 2

[solution n°3 p.27]

Les utilisateurs souhaiteraient aussi pouvoir rechercher un jeu par leur description, en saisissant un mot qu'elle contient. Rédigez la requête préparée permettant d'implémenter cette fonctionnalité en utilisant des marqueurs interrogatifs.

VI. PDOStatement

¹ <https://www.php.net/manual/fr/pdo.prepared-statements.php>

Objectifs

- Découvrir la classe `PDOStatement`
- Préparer une requête
- Donner des valeurs aux marqueurs
- Lancer une requête préparée

Mise en situation

Maintenant que nous savons former des requêtes préparées, il faut les donner à PDO et fournir les valeurs à injecter dans nos marqueurs. Pour cela, nous allons manipuler un autre objet issu de la librairie PDO : le `PDOStatement`.

Méthode Préparer une requête

Pour préparer une requête, il faut utiliser la méthode `prepare` de notre objet PDO. Elle prend en paramètre la requête préparée et retourne un objet de type `PDOStatement`.

Cette méthode n'exécute pas la requête : elle ne fait que la garder en mémoire pour pouvoir la manipuler via le `PDOStatement`.

Définition PDOStatement

Un objet `PDOStatement` est la représentation d'une requête et permet de réaliser toutes les actions que l'on fait habituellement avec une requête : l'exécuter, récupérer le résultat, renseigner les valeurs des marqueurs, récupérer les erreurs, etc.

On peut voir la classe `PDO` comme une Factory qui va permettre d'instancier des `PDOStatement`, un par requête que l'on va vouloir exécuter.

Remarque

Factory est un design pattern permettant de mieux structurer des classes. Le but étant d'obtenir une classe qui va se charger de créer plusieurs objets en utilisant une rédaction plus simple à base de méthodes statiques qui retourneront des instances.

Méthode Renseigner les valeurs des marqueurs

L'objet `PDOStatement` va nous permettre de renseigner les valeurs de nos marqueurs. Pour cela, il met deux méthodes à notre disposition : `bindValue` et `bindParam`.

- Ces deux méthodes, très similaires, prennent deux paramètres obligatoires : le marqueur à remplacer et la valeur à lui attribuer. Le premier paramètre va différer selon le type de marqueur utilisé : pour les marqueurs nommés, il faut passer le nom du marqueur, tandis que pour des marqueurs interrogatifs, il faut passer leur position.
- La différence entre les deux méthodes est subtile, mais à son importance : `bindValue` va permettre de lier une valeur à un paramètre, tandis que `bindParam` va lier une variable par référence à un paramètre. Ainsi, avec `bindParam`, si le paramètre est modifié dans la requête SQL (par exemple, lors d'appels à des procédures stockées), alors la variable PHP sera également modifiée.
- Le troisième paramètre, facultatif mais recommandé, est le type de variable à choisir entre `PDO::PARAM_NULL`, `PDO::PARAM_BOOL`, `PDO::PARAM_INT` et `PDO::PARAM_STR`. Cela permet d'indiquer à PDO comment formater les valeurs (par exemple, les mettre entre apostrophes dans le cas d'une chaîne de caractères). Par défaut, PDO les considère comme des chaînes de caractères.

Dans les deux cas suivants, nous retrouvons des marqueurs nommés. L'un est utilisé par `bindValue`, l'autre par `bindParam`. En effet, dans le premier exemple ci-dessous, lors de l'exécution de la requête, la valeur de `$username` prise en compte sera 'Jean' car elle est rattachée au marqueur de la requête.

```
1 <?php
2 $username = 'Jean';
3 $password = '1234';
4 // Marqueurs nommés et bindValue:
5 $statement = $pdo->prepare("SELECT * FROM users WHERE `username`=:username AND
6 `pass`=:pass");
7 $statement->bindValue(':username', $username, PDO::PARAM_STR);
8 $statement->bindValue(':pass', $password, PDO::PARAM_STR);
9 $username = 'Pierre';
10 $statement->execute();
11 $user = $requete->fetchAll();
```

Dans le cas contraire qui est le suivant, `$username` prendra comme valeur 'Pierre' lors de l'exécution de la requête car tant que celle-ci n'a pas aboutie, la valeur injectée pourra être modifiée.

```
1 <?php
2 $username = 'Jean';
3 $password = '1234';
4 // Marqueurs nommés et bindParam:
5 $statement = $pdo->prepare("SELECT * FROM users WHERE `username`=:username AND
6 `pass`=:pass");
7 $statement->bindParam(':username', $username, PDO::PARAM_STR);
8 $statement->bindParam(':pass', $password, PDO::PARAM_STR);
9 $username = 'Pierre';
10 $statement->execute();
11 $user = $requete->fetchAll();
```

Il s'agit du même fonctionnement avec les marqueurs interrogatifs et les seules données changeantes seront :

```
1 // Marqueurs interrogatifs et bindParam:
2 $statement = $pdo->prepare("SELECT * FROM users WHERE `username`= ? AND `pass`= ?");
3 $statement->bindParam(1, $username, PDO::PARAM_STR);
4 $statement->bindParam(2, $password, PDO::PARAM_STR);
```

Il est à noter que `bindParam` doit obligatoirement avoir une variable en paramètre, puisqu'elle la lie par référence.

Attention

Contrairement à ce à quoi on commence à être habitué en informatique, la position des marqueurs interrogatifs commence à 1 et non à 0.

Remarque

Les cas d'utilisation de `bindParam` sont très spécifiques. Dans la majorité des cas, `bindValue` est amplement suffisant et est plus pratique à manipuler.

Exécuter la requête

Une fois les paramètres renseignés, nous allons pouvoir exécuter notre requête préparée. Pour cela, il suffit d'utiliser la méthode `execute` de notre objet `PDOStatement`. Cette méthode retourne un booléen indiquant si la requête s'est bien déroulée ou non.

En cas d'erreur, la méthode `errorInfo` permet de récupérer des informations sur l'erreur. Elle retourne un tableau, dont le premier élément est le SQLSTATE (c'est-à-dire le code d'erreur SQL), le second élément est le code d'erreur du driver et le troisième est le message d'erreur.

Exemple

```
1 <?php
2 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
3 // Faute de frappe volontaire dans la requête pour tester l'erreur
4 $statement = $pdo->prepare('ELECT * FROM users WHERE name LIKE :name');
5 $statement->bindValue(':name', 'a%', PDO::PARAM_STR);
6 if ($statement->execute()) {
7     // La requête s'est bien déroulée
8 } else {
9     $errorInfo = $statement->errorInfo();
10    echo 'SQLSTATE : '.$errorInfo[0].'\n';
11    echo 'Erreur du driver : '.$errorInfo[1].'\n';
12    echo 'Message : '.$errorInfo[2];
13 }
```

```
SQLSTATE : 42000
Erreur du driver : 1064
Message : You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near 'ELECT * FROM users WHERE username LIKE 'a%' at line 1
```

Attention

À l'image des `PDOException`, les informations remontées par `errorInfo` sont susceptibles de comporter des informations sensibles concernant votre base de données. Il est fortement recommandé de logger ces erreurs plutôt que de les afficher directement sur la page.

```
1 <?php
2 // Message d'erreur à sauvegarder
3 $errorMessage = "Ceci est un message d'erreur!";
4
5 // Chemin du fichier log où les erreurs doivent être sauvegardées
6 $logFile = "./errors.log";
7
8 // Enregistrement du message d'erreur dans le fichier log
9 error_log($errorMessage, 3, $logFile);
10 ?>
```

Complément

Il est également possible de passer un tableau contenant les valeurs de nos marqueurs en paramètre d'`execute` : PDO fera automatiquement des `bindValue` en utilisant les valeurs fournies. Dans le cas de marqueurs nommés, chaque clé doit correspondre à un nom de paramètre. En revanche, pour les marqueurs interrogatifs, seul l'ordre des valeurs dans le tableau compte.

```
1 <?php
2 // Marqueurs nommés
3 $statement = $pdo->prepare('SELECT * FROM users WHERE name LIKE :name');
4 if ($statement->execute([':name' => 'j%'])) {
5     // La requête s'est bien déroulée
6 }
7
8 // Marqueurs interrogatifs
9 $statement = $pdo->prepare('SELECT * FROM users WHERE name LIKE ?');
10 if ($statement->execute(['j%'])) {
```

```
11 // La requête s'est bien déroulée
12 }
```

Cette méthode, bien que plus rapide, ne permet pas de définir le type de chaque valeur, toutes considérées comme des PDO : : PARAM_STR. Il faut donc être vigilant quant à son utilisation.

Fondamental Exécuter plusieurs requêtes

En plus des avantages de sécurité des requêtes préparées, celles-ci sont aussi très performantes pour exécuter des requêtes similaires. En effet, lorsqu'un SGBD reçoit une requête, il va d'abord l'analyser pour optimiser la requête avant de l'exécuter.

Cette phase d'optimisation ne dépend pas des valeurs de nos filtres : elle se base uniquement sur les tables et les champs qui sont impliqués dans la requête. Ainsi, utiliser la méthode `query` pour lancer plusieurs fois la même requête, mais avec des valeurs différentes, va lancer cette phase d'optimisation pour chaque requête, ce qui n'est pas optimal.

En revanche, préparer une requête permet au SGBD de ne l'analyser qu'une seule fois : chaque nouvelle exécution utilisera la même optimisation.

Méthode Créer des procédures stockées

Les procédures stockées sont utilisables depuis la version 5 de MySQL. À la différence des requêtes préparées, ces procédures sont stockées durablement et font partie de la base de données dans laquelle elles sont enregistrées.

Elles sont utilisées en administration de base de données afin d'exécuter une série d'instructions SQL désignées par un nom.

Une fois celles-ci créées il est possible de les appeler par leurs noms. Nous allons maintenant apprendre et comprendre comment créer une procédure stockée. Tout d'abord, pour créer une procédure, la commande à exécuter est `CREATE PROCEDURE` suivi du nom que l'on souhaite lui donner. Suite à cela, on ajoutera des parenthèses **OBLIGATOIRES** qui nous permettront de définir les futurs paramètres de la procédure.

```
1 CREATE PROCEDURE nom_de_la_procedure()
```

Venons en au corps de la procédure : c'est dans ce corps que l'on rentre le contenu de la procédure, il s'agit de ce qui sera exécuté. Il peut être question d'une requête ou d'un bloc d'instruction.

```
1 CREATE PROCEDURE nom_de_la_procedure()
2 corps de la procédure;
```

Afin de délimiter un bloc d'instructions, on utilise les termes `BEGIN` et `END`.

```
1 CREATE PROCEDURE nom_de_la_procedure()
2 BEGIN
3     --instructions;
4 END;
5
```

En revanche, ce simple aperçu de code précédent risque de nous poser problème car dans le cas où nous remplacerions votre commentaire par une série d'instructions, il y aurait un conflit. En effet, le premier ";" pourrait porter atteinte au bon déroulement de l'exécution du code car il provoque la fin de l'exécution avant "END;".

Afin de remédier à ce problème, il existe ce que l'on appelle un délimiteur. Par défaut, sa valeur prendra le caractère ";" soit, le caractère qui permet de délimiter les instructions.

Cependant, il est possible de modifier celui-ci. Pour changer le délimiteur, vous pouvez utiliser la commande `DELIMITER`.

Les deux délimiteurs les plus utilisés sont les suivants :

```
1 DELIMITER |
2 DELIMITER \\\
```

Attention

Vous pouvez utiliser le ou les caractères de votre choix comme délimiteur mais veillez à ce que votre futur délimiteur ne soit pas un caractère trop classique car celui-ci risque d'être réutilisé par la suite.

Remarque

Pour déclencher l'exécution du bloc d'instructions, vous devrez utiliser le mot-clé CALL à la suite duquel vous placerez le nom de la procédure appelée, suivie de parenthèses.

```
1 CALL nom_de_la_procedure()
```

Exemple

Imaginons que l'on veuille sélectionner tous les pays d'Europe :

- On change le délimiteur de sorte à ne pas provoquer d'erreur,
- On crée une procédure du nom de *afficher_pays*,
- On déclare le début de l'instruction ainsi que sa fin en n'oubliant pas d'adapter le délimiteur,
- Enfin, on rentre nos instructions dans le corps de la procédure. Ici on cherche à sélectionner le pays ainsi que son id dans la table Europe.DELIMITER | -- On change le délimiteur

```
1 DELIMITER | -- On change le délimiteur
2 CREATE PROCEDURE afficher_pays()
3 BEGIN
4     SELECT id, pays
5     FROM Europe;
6 END |
7
```

Syntaxe À retenir

- Pour préparer une requête, il faut utiliser la méthode `prepare` de l'objet PDO. Elle retourne un `PDOStatement`, qui représente une requête. Pour donner une valeur aux marqueurs, il faut utiliser les méthodes `bindValue`, pour lier une valeur à un marqueur, ou `bindParam`, pour lier une variable par référence. Une fois les valeurs renseignées, il faut exécuter la requête avec la méthode `execute`.
- En plus du formatage automatique des données par PDO, préparer une requête permet d'optimiser l'exécution de requêtes multiples.

Complément

Les requêtes préparées¹

¹ <https://www.php.net/manual/fr/pdo.prepared-statements.php>

VII. Exercice : Appliquez la notion

Question 1

[solution n°4 p.27]

L'administrateur de notre site de tests de jeux doit pouvoir supprimer un jeu de son choix. Pour cela, créez une page permettant de récupérer l'identifiant d'un jeu depuis un paramètre passé dans l'URL et de supprimer le jeu associé. Pour rappel, la structure de la table des jeux est la suivante :

```
1 CREATE TABLE games(  
2     id INTEGER(11) PRIMARY KEY AUTO_INCREMENT,  
3     name VARCHAR(500),  
4     description VARCHAR(500)  
5 );  
6 INSERT INTO games(name, description) VALUES  
7 ('Super Moria World', 'Parcourez le monde pour jeter une tortue dans la lave !'),  
8 ('The legend of Zebda', 'Une épopée musicale où vous devez vaincre les forces du mal.');
```

En cas de problème lors de l'exécution de la requête, affichez simplement le message d'erreur et en cas de succès, un message de succès.

On peut noter que, pour simplifier l'exercice, nous passons l'ID dans l'URL. Mais, dans la pratique, il serait grandement préférable de passer l'identifiant via un formulaire.

Indice :

N'oubliez pas de préciser le type du `bindValue`.

Question 2

[solution n°5 p.27]

Améliorons la gestion des erreurs de la requête. Créez une table en base de données qui va recevoir les informations de l'erreur :

```
1 CREATE TABLE errors(  
2     id INTEGER(11) PRIMARY KEY AUTO_INCREMENT,  
3     state VARCHAR(10),  
4     driverError VARCHAR(50),  
5     message VARCHAR(500)  
6 );
```

Modifiez votre gestion d'erreurs pour insérer toutes les informations des erreurs dans cette table.

Indice :

La méthode `errorInfo` retourne un tableau indexé contenant les différentes informations, toutes stockées en tant que chaînes de caractères. C'est dans ces cas-là que les marqueurs interrogatifs et l'injection de valeurs dans la méthode `execute` sont les plus efficaces.

VIII. Récupérer le résultat

Objectifs

- Récupérer le résultat de notre requête préparée
- Découvrir les différents modes de récupération

Mise en situation

La requête préparée étant exécutée, il faut maintenant récupérer les résultats. Jusqu'à présent, nous n'avons utilisé que le mode `PDO::FETCH_ASSOC` avec la méthode `query`, mais il est temps de voir d'autres manières de gérer nos données.

Méthode La méthode fetch

Pour récupérer les résultats après avoir exécuté une requête préparée, il faut utiliser la méthode `fetch` de notre objet `PDOStatement`, en lui précisant le mode de récupération souhaité.

`fetch` est une méthode particulière : elle permet de récupérer la prochaine ligne de résultats, ou `false` en cas d'erreur (ou s'il n'y a plus de résultat à retourner). Nous allons donc devoir boucler sur `fetch` en récupérant chaque ligne :

```
1 <?php
2 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
3 $statement = $pdo->prepare('SELECT * FROM users WHERE name LIKE :name');
4 $statement->bindValue(':name', 'a%');
5 if ($statement->execute()) {
6     while ($user = $statement->fetch(PDO::FETCH_ASSOC)) {
7         echo '<pre>';
8         print_r($user);
9         echo '</pre>';
10    }
11 }
```

Remarque

En réalité, la méthode `query`, vue précédemment, retourne également un objet de type `PDOStatement`. Un des avantages de la classe `PDOStatement` est qu'elle implémente l'interface `Traversable`, qui permet de définir le comportement d'un objet lorsque l'on essaye de boucler dessus. Ainsi, utiliser un `foreach` sur un `PDOStatement` permet également de boucler sur les résultats de la requête. Dans ce cas, il faut utiliser la méthode `setFetchMode` pour définir le mode de récupération.

Récupérer un tableau indexé

`fetch` est capable de retourner les résultats sous différentes formes : nous connaissons `PDO::FETCH_ASSOC`, qui retourne les résultats sous forme de tableaux associatifs, mais il en existe trois principaux autres.

- `PDO::FETCH_NUM` permet de récupérer les résultats sous forme de tableaux indexés : les clés du tableau ne seront plus le nom des colonnes, mais simplement leur position dans la requête `SELECT`. Bien que beaucoup moins utilisé, il peut parfois être utile pour créer des fonctions qui s'adaptent à plusieurs tables. De plus, étant donné que `PDO::FETCH_NUM` ne se base pas sur les noms de colonnes, il n'y a aucun risque d'écrasement de clés si deux colonnes ont le même nom. Les alias ne sont donc pas utiles avec ce mode de récupération.

Exemple

Avec le `PDO::FETCH_NUM`, les utilisateurs seront sous la forme :

```
Array
(
    [0] => 1
    [1] => john
    [2] => 109f4b3c50d7b0df729d299bc6f8e9ef9066971f
    [3] => john@doe.com
    [4] => 1
)
```


Leur manipulation en PHP devra utiliser des indices numériques :

```
1 <?php
2 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
3 $statement = $pdo->prepare('SELECT * FROM users WHERE name LIKE :name');
4 $statement->bindValue(':name', 'j%');
5 if ($statement->execute()) {
6     while ($user = $statement->fetch(PDO::FETCH_NUM)) {
7         echo $user[1].<br>; // Affiche le nom
8     }
9 }
```

Remarque

Il est possible de combiner `FETCH_ASSOC` et `FETCH_NUM` en utilisant `FETCH_BOTH`. Le tableau final aura ainsi des clés numériques et des noms de colonnes.

```
array(10) {
    ["id"]=>
    string(1) "1"
    [0]=>
    string(1) "1"
    ["name"]=>
    string(4) "john"
    [1]=>
    string(4) "john"
    ["password"]=>
    string(40) "109f4b3c50d7b0df729d299bc6f8e9ef9066971f"
    [2]=>
    string(40) "109f4b3c50d7b0df729d299bc6f8e9ef9066971f"
    ["email"]=>
    string(12) "john@doe.com"
    [3]=>
    string(12) "john@doe.com"
    ["groupId"]=>
    string(1) "1"
    [4]=>
    string(1) "1"
}
```

Récupérer un objet standard

- `PDO::FETCH_OBJ` permet de récupérer les résultats sous la forme d'un objet standard, instance de `stdClass`, dont chaque propriété sera une colonne de la table.

La `stdClass` est une classe générique que l'on peut voir comme une implémentation objet d'un tableau associatif : elle n'a pas de méthode et ses propriétés peuvent être définies dynamiquement.

Exemple

Avec le `PDO::FETCH_OBJ`, les utilisateurs seront sous la forme :

```
object(stdClass)#4 (5) {
    ["id"]=>
    string(1) "1"
    ["name"]=>
    string(4) "john"
    ["password"]=>
    string(40) "109f4b3c50d7b0df729d299bc6f8e9ef9066971f"
    ["email"]=>
    string(12) "john@doe.com"
    ["groupId"]=>
    string(1) "1"
}
```

Leur manipulation en PHP devra utiliser des accesseurs de propriété :

```
1 <?php
2 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
3 $statement = $pdo->prepare('SELECT * FROM users WHERE name LIKE :name');
4 $statement->bindValue(':name', 'j%');
5 if ($statement->execute()) {
6     while ($user = $statement->fetch(PDO::FETCH_OBJ)) {
7         echo $user->name.'<br>'; // Affiche le nom
8     }
9 }
```

Récupérer un objet de l'application

- Enfin, `PDO::FETCH_CLASS` est un peu particulier car il nécessite de renseigner le mode de récupération grâce à la méthode `setFetchMode` plutôt que dans la méthode `fetch`. Dans ce cas, `fetch` n'aura pas de paramètres. Ce mode permet d'instancier une classe fournie en paramètre plutôt qu'une `stdClass` : les propriétés correspondant aux noms des colonnes seront alimentées par les valeurs de la ligne.

Bien qu'un peu plus complexe à mettre en place, c'est de loin la meilleure option, puisqu'elle permet de profiter de tous les avantages de la programmation orientée objet.

Exemple

Si on définit une classe `User` :

```
1 <?php
2
3 class User
4 {
5     private string $id;
6     private string $name;
7     private string $password;
8     private string $email;
9     private int $groupId;
10
11     public function getDisplayedName()
12     {
13         return $this->name.'<br>';
14     }
15 }
```

PDO peut créer automatiquement des instances de cette classe :

```
1 <?php
2
3 require_once 'User.php';
4 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
5 $statement = $pdo->prepare('SELECT * FROM users WHERE name LIKE :name');
6 $statement->bindValue(':name', 'j%');
7 $statement->setFetchMode(PDO::FETCH_CLASS, 'User'); // On appelle setFetchMode en lui donnant
   le nom de la classe
8 if ($statement->execute()) {
9     while ($user = $statement->fetch()) { // fetch n'a pas de paramètre : on a défini le mode
   au dessus
10         echo $user->getDisplayedName(); // $user est un objet User
11     }
12 }
```

Conseil

Dans le cas d'un `PDO::FETCH_CLASS`, il est possible d'utiliser la méthode `fetchObject` au lieu de la méthode `fetch` pour ne pas avoir à appeler `setFetchMode`. Cette méthode prend en paramètre la classe à instancier :

```
1 <?php
2 require_once 'User.php';
3 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
4 $statement = $pdo->prepare('SELECT * FROM users WHERE name LIKE :name');
5 $statement->bindValue(':name', 'j%');
6 if ($statement->execute()) {
7     while ($user = $statement->fetchObject('User')) { // On utilise fetchObject
8         echo $user->getDisplayedName();
9     }
10 }
```

Récupérer tous les résultats

Si l'on ne souhaite pas récupérer les résultats ligne par ligne, mais **tout récupérer** d'un coup, il est possible d'utiliser la méthode `fetchAll`. Les paramètres sont identiques à ceux de `fetch`, mais `fetchAll` retourne un tableau contenant tous les résultats, au format voulu.

```
1 <?php
2 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
3 $statement = $pdo->prepare('SELECT * FROM users WHERE name LIKE :name');
4 $statement->bindValue(':name', 'j%');
5 if ($statement->execute()) {
6     $users = $statement->fetchAll(PDO::FETCH_OBJ); // $users contient un tableau d'objets
   StdClass
7 }
```

Attention

L'intérêt de récupérer les données ligne par ligne est d'utiliser peu d'espace mémoire : seules les données d'une seule ligne sont stockées et elles sont remplacées par la ligne d'après.

En utilisant `fetchAll`, l'intégralité des données retournées par la requête seront stockées en mémoire, ce qui peut provoquer des lenteurs ou des erreurs. N'utilisez cette méthode que si vous en avez vraiment besoin et que vous maîtrisez le nombre de données retournées par la requête.

Syntaxe À retenir

Pour récupérer les résultats ligne par ligne, il faut utiliser la méthode `fetch` en lui précisant le mode de récupération. Il en existe 4 principaux :

- `PDO::FETCH_ASSOC` qui retourne les résultats sous forme de tableau associatif
- `PDO::FETCH_NUM` qui retourne les résultats sous forme de tableau indexé
- `PDO::FETCH_OBJ` qui retourne les résultats sous forme d'objet standard `stdClass`
- `PDO::FETCH_CLASS` qui retourne les résultats sous forme d'instance d'une classe de l'application. Ce mode nécessite l'appel à `setFetchMode` en amont, mais cela peut être évité en utilisant la méthode `fetchObject`.

Il est également possible de récupérer tous les résultats, plutôt que ligne par ligne, via la méthode `fetchAll`, mais ce n'est pas recommandé : tout le retour de la requête sera alors stocké en mémoire.

Complément

La méthode `fetch`¹

La méthode `fetchAll`²

IX. Exercice : Appliquez la notion

Question

[solution n°6 p.28]

Créez une classe `Game` permettant de représenter un jeu stocké en base de données. Cette classe devra avoir une méthode permettant d'afficher la concaténation du nom du jeu et de sa description.

Puis, en vous aidant des exercices précédents, modifiez la page de la liste des jeux en utilisant une requête préparée qui récupère les jeux. Donnez la possibilité aux utilisateurs de renseigner un nom en paramètre de l'URL pour ne récupérer que les jeux qui commencent par ce nom.

En cas d'erreur, affichez le message d'erreur.

Pour rappel, la table permettant de gérer les jeux est sous la forme :

```
1 CREATE TABLE games(
2     id INTEGER(11) PRIMARY KEY AUTO_INCREMENT,
3     name VARCHAR(500),
4     description VARCHAR(500)
5 );
6 INSERT INTO games(name, description) VALUES
7 ('Super Moria World', 'Parcourez le monde pour jeter une tortue dans la lave !'),
8 ('The legend of Zebda', 'Une épopée musicale où vous devez vaincre les forces du mal, Manau a
    Manau');
```

X. La pagination

Objectifs

- Séparer notre liste en plusieurs pages
- Trouver le nombre de pages

1 <https://www.php.net/manual/fr/pdostatement.fetch.php>

2 <https://www.php.net/manual/fr/pdostatement.fetchall.php>

Mise en situation

Notre liste d'utilisateurs est presque prête, il ne reste plus qu'un dernier problème à gérer. Pour le moment, nous récupérons l'intégralité des utilisateurs de notre base, ce qui peut représenter une grande quantité de données à afficher. En plus de ne pas être très ergonomique pour l'utilisateur, cela peut poser des problèmes de performance : le temps de chargement de la page étant proportionnel au nombre d'utilisateurs inscrits, combien de temps notre application va mettre pour afficher 10 000, 100 000, voire 1 000 000 d'utilisateurs ?

Une solution simple existe pour pallier ce problème : la pagination.

Méthode Créer un système de pagination

Plutôt que de charger l'intégralité de nos utilisateurs, nous n'allons en afficher qu'un faible nombre avec la possibilité de changer de page pour charger ceux qui suivent. Cela permet de maîtriser la quantité de données qui va être chargée depuis la base : elle n'est plus proportionnelle au nombre d'utilisateurs, mais est maintenant un nombre fixe que l'on peut modifier selon les performances de l'application.

Pour limiter le nombre d'utilisateurs chargés, nous allons utiliser la clause SQL `LIMIT`, qui permet d'indiquer le nombre maximum de lignes à récupérer et le numéro de la ligne à partir de laquelle les données doivent être retournées. Par exemple, `LIMIT 5, 15` récupérera 15 lignes à partir de la sixième, donc les lignes 6 à 20.

Exemple

Pour récupérer les utilisateurs d'une certaine page, il faudra donc jouer sur la première valeur du `LIMIT`. Si nous souhaitons afficher 10 utilisateurs par page, alors il faudra afficher les utilisateurs 1 – 10 sur la page 1, puis 11 – 20 sur la page 2, et 21 – 30 sur la page 3, etc. Cela signifie que cette valeur devra être égale au *nombre d'utilisateurs par page * (numéro de page – 1)*.

```
1 <?php
2
3 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
4 $statement = $pdo->prepare('SELECT * FROM users LIMIT :start, 10'); // La deuxième valeur du
LIMIT est 10
5 $statement->bindValue('start', 10 * ($_GET['page'] - 1), PDO::PARAM_INT); // On calcule la
première valeur dans le bindValue. Ici, le numéro de page est passé dans l'URL.
6 if ($statement->execute()) {
7     while ($user = $statement->fetch(PDO::FETCH_ASSOC)) {
8         echo $user['name'].'<br>';
9     }
10 }
```

Attention

Si le type de données, qui est le troisième paramètre de la fonction `bindValue`, n'est pas précisé alors PDO va considérer que c'est une chaîne de caractères. Ainsi, dans la requête finale, même si c'est un nombre qui est fourni en tant que valeur du marqueur, il sera mis entre apostrophes par PDO.

Cela a pour effet de potentiellement ralentir la requête : si les colonnes sont définies comme des index numériques, alors certains SGBD ne pourront pas profiter de l'index.

Pire, dans certains cas, la requête pourrait échouer. En effet, si on utilise un marqueur pour gérer la valeur d'une clause `LIMIT`, comme c'est le cas dans l'exemple ci-dessus, alors la traiter comme une chaîne de caractères générerait la requête suivante :

```
SELECT * FROM users LIMIT '10', 10;
```

Or, donner une chaîne de caractères à une clause `LIMIT` provoque une erreur :

```
Error: ER_PARSE_ERROR: You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use near
''10'' at line 1
```

Il faut donc être vigilant au type des données au moment du *bind*. Dans le doute, il est préférable de toujours fournir un troisième paramètre, quel que soit le type.

Récupérer le nombre de pages

Nous savons afficher les utilisateurs d'une page précise, mais il nous faut maintenant pouvoir changer de page. Pour cela, nous allons afficher toutes les pages disponibles, avec un lien permettant d'y accéder. Nous devons donc connaître le nombre de pages, puis compter de la page 1 à ce nombre.

Pour connaître le nombre de pages, il suffit de récupérer le nombre d'utilisateurs dans notre base de données grâce à la fonction `COUNT` de SQL. Une fois le nombre total d'utilisateurs connu, le nombre de page est égal au *nombre d'utilisateurs total / nombre d'utilisateurs par page*, le tout arrondi au supérieur (calculé grâce à la fonction native de PHP `ceil`).

```
1 <?php
2 $statement = $pdo->prepare('SELECT COUNT(*) AS totalUsers FROM users');
3 if ($statement->execute()) {
4     $totalUsers = $statement->fetch(PDO::FETCH_ASSOC); // Ici, nous savons que nous n'allons
    recevoir qu'une seule ligne, donc il est possible d'appeler fetch directement
5     for ($i = 1; $i <= ceil($totalUsers['totalUsers'] / 10); $i++) {
6         echo '<a href="?page=' . $i . '>' . $i . '</a> - ';
7     }
8 }
```

Syntaxe À retenir

- L'instruction `LIMIT` permet de ne récupérer qu'un certain nombre d'éléments à partir d'une certaine position. On l'utilise pour la pagination.
- Pour récupérer les éléments d'une certaine page, il faut commencer par récupérer l'élément à la position *nombre d'utilisateurs par page * (numéro de page - 1)*.
- Le nombre de page total est *nombre d'utilisateurs total / nombre d'utilisateurs par page*, arrondi au supérieur.

Complément

`ceil`¹

XI. Exercice : Appliquez la notion

Question

[solution n°7 p.28]

Modifiez le script d'affichage de la liste des jeux de la question précédente, pour n'afficher les jeux que 5 par 5, selon un numéro de page passé en paramètre. Pour rappel, voici la structure de la table des jeux et une liste de jeux à insérer, qui doivent être affichés sur deux pages :

```
1 CREATE TABLE games(
2     id INTEGER(11) PRIMARY KEY AUTO_INCREMENT,
3     name VARCHAR(500),
4     description VARCHAR(500)
5 );
6 INSERT INTO games(name, description) VALUES
7 ('Super Moria World', 'Parcourez le monde pour jeter une tortue dans la lave !'),
8 ('Super Moria World 2 : les deux Boos', 'Une nouvelle aventure vous attend.'),
9 ('Super Moria World 3 : Le retour du Koopa', 'Le dernier volet de la trilogie.'),
```

¹ <https://www.php.net/manual/fr/function.ceil.php>

```
10 ('The legend of Zebda', 'Une épopée musicale où vous devez vaincre les forces du mal.'),  
11 ('The legend of Zebda : Gojira''s Mask', 'Empêchez le ciel de vous tomber sur la tête !'),  
12 ('The legend of Zebda : Queen''s awakening', 'Le spectacle doit continuer');
```

Indice :

Il est critique, ici, de bien spécifier le type dans le `bindValue`.

XII. Essentiel

XIII. Auto-évaluation

A. Exercice final

Exercice 1

[solution n°8 p.29]

Exercice

À quoi sert la méthode `query` ?

- ☐ Se connecter à la base de données
- ☐ Lancer une requête et récupérer le résultat
- ☐ Lancer une requête et récupérer le nombre de lignes affectées

Exercice

Lorsque l'on récupère le résultat d'une requête `SELECT` en PHP avec le mode `PDO::FETCH_ASSOC`, comment est déterminé le nom de chaque clé ?

- ☐ C'est la position de chaque colonne dans la requête
- ☐ C'est le nom de chaque colonne des tables concernées par la requête
- ☐ C'est le nom de chaque colonne de la table virtuelle retournée par la requête

Exercice

Soit la table **garages**, ayant pour champs `id`, `name` et `address` et la table **cars**, ayant pour champs `id`, `garageId`, `name` et `brand`. Combien de colonnes est-ce que le résultat de la requête `SELECT * FROM garages JOIN cars ON garages.id = cars.garageId` possédera-t-il en utilisant le mode `PDO::FETCH_ASSOC` ?

- ☐ 7
- ☐ 6
- ☐ 5
- ☐ 4
- ☐ 3

Exercice

Pour les mêmes tables et la même requête que la question précédente, combien de champs possèdera le tableau de résultats récupérés en utilisant le mode `PDO::FETCH_NUM` ?

- ☐ 7
- ☐ 6
- ☐ 5
- ☐ 4
- ☐ 3

Exercice

Parmi ces requêtes préparées, lesquelles sont valides ?

- ☐ `SELECT * FROM cars WHERE brand LIKE :search`
- ☐ `SELECT * FROM cars WHERE brand LIKE :search%`
- ☐ `SELECT * FROM cars WHERE brand LIKE ?`
- ☐ `SELECT * FROM :table WHERE brand LIKE :search`
- ☐ `SELECT * FROM cars WHERE $field LIKE :search`

Exercice

Qu'est-ce qu'un objet `PDOStatement` ?

- ☐ La représentation d'une connexion à la base de données
- ☐ La représentation d'une requête
- ☐ La représentation d'une ligne de données

Exercice

Le code ci-dessous est valide.

```
1 <?php
2
3 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
4 $statement = $pdo->prepare('SELECT * FROM cars WHERE brand LIKE :search');
5 $statement->bindParam(':search', 'audi%', PDO::PARAM_STR);
6 if ($statement->execute()) {
7     while ($car = $statement->fetch(PDO::FETCH_ASSOC)) {
8         echo $car['name'];
9     }
10 }
```

- ☐ Vrai
- ☐ Faux

Exercice

À quoi sert le mode de récupération PDO : `:FETCH_OBJ` ?

- ☐ Récupérer les données sous forme d'instance de `stdClass`, la classe standard de PHP
- ☐ Récupérer les données sous forme d'instance d'objet de notre application : il faut alors lui donner le nom de la classe à instancier
- ☐ Récupérer les données sous forme d'un mélange entre le tableau associatif et indexé

Exercice

Combien de résultats va retourner la requête `SELECT * FROM cars LIMIT 5, 10` ?

- ☐ Au moins 5
- ☐ Au plus 5
- ☐ Au moins 10
- ☐ Au plus 10

Exercice

Comment faire pour récupérer les résultats de la page X de mon site, sachant qu'il y a 12 voitures par page ?

- ☐ `SELECT * FROM cars LIMIT X, 12`
- ☐ `SELECT * FROM cars LIMIT X * 12, 12`
- ☐ `SELECT * FROM cars LIMIT (X - 1) * 12, 12`
- ☐ `SELECT * FROM cars LIMIT (X + 1) * 12, 12`

B. Exercice : Défi

Dans cet exercice, vous allez créer votre propre fil d'actualité, à la façon de Twitter ou Facebook ! Le but est d'avoir un formulaire (composé d'un seul `textarea` et d'un bouton de soumission) permettant de poster des messages courts.

Question 1

[solution n°9 p.32]

Dans votre application, un message est composé d'un contenu textuel et d'une date (avec l'heure).

Créez une table Messages permettant de gérer ces informations, en ajoutant les champs nécessaires à son optimisation. De plus, étant donné qu'à chaque création de message, la date renseignée sera la date actuelle, mettez cette valeur en valeur par défaut.

Créez également une classe PHP `Message` représentant un message dans la base de données. Vous allez avoir besoin d'afficher leur contenu et leur date, donc ajoutez les accesseurs nécessaires.

Indice :

Les dates sont gérées en tant que chaînes de caractères par PDO, donc le type du champ date de la classe `Message` devra être string.

Question 2

[solution n°10 p.32]

Créez un formulaire HTML composé d'un `textarea` et un bouton de soumission. Le formulaire devra pointer sur la même page en `POST`. Pour respecter le modèle MVC, mettez le code HTML dans un fichier à part, qui sera inclus dans votre script principal (qui fera office de contrôleur).

Indice :

N'oubliez pas de donner un attribut `name` au `textarea` afin de pouvoir le manipuler en PHP.

Question 3

[solution n°11 p.33]

Créez une classe `MessageManager`, qui va vous permettre de manipuler les messages en base de données. Cette classe devra posséder une instance de PDO en propriété, qui sera alimentée par injection de dépendances.

Dans un premier temps, vous allez gérer l'ajout de messages : ajoutez au manager une méthode `addMessage(string $content) : bool` permettant d'insérer un message ayant le contenu passé en paramètre en base de données. Cette méthode doit retourner « vrai » si l'insertion a bien été effectuée et « faux » sinon.

Ensuite, modifiez votre contrôleur de manière à ajouter un message à la soumission du formulaire.

Enfin, ajoutez la phrase « *Votre message a bien été ajouté !* » dans la vue. Ce message ne doit apparaître que si la requête a bien été exécutée. Attention : respectez bien le découpage MVC.

Indice :

« Par injection de dépendances » signifie que l'objet PDO qui sera utilisé par le manager sera renseigné dans le constructeur, au moment de la création de l'objet. L'objet PDO sera créé dans le script principal et injecté dans votre manager.

Indice :

Pour savoir si le formulaire a été soumis ou non, il est possible d'utiliser `if (!empty($_POST['content']))`. Cela permet également de s'assurer que le message n'est pas vide.

Question 4

[solution n°12 p.34]

- Affichez maintenant les messages qui ont été créés : dans votre `MessageManager`, ajoutez une méthode `getMessages(int $page)` qui retourne la liste d'objets `Message` à afficher pour la page donnée en paramètre. Pour cet exercice, vous allez afficher les messages 10 par 10. Étant donné que vous contrôlez la quantité de données renvoyées et donc la quantité de mémoire utilisée, la méthode `fetchAll` pourra être utilisée pour simplifier le code.
- Créez ensuite une nouvelle vue qui va s'occuper d'afficher un message, avec son contenu et sa date. Cette nouvelle vue devra être appelée dans la vue principale.
- Enfin, modifiez le contrôleur de manière à charger les messages d'une page, dont le numéro est passé en paramètre dans l'URL et à les donner à la vue. Si le paramètre n'est pas présent dans l'URL, alors la valeur par défaut est 1.

Solutions des exercices

p. 6 Solution n°1

```

1 <?php
2
3 try {
4     $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
5     foreach ($pdo->query('SELECT name, description FROM games', PDO::FETCH_ASSOC) as $user) {
6         echo $user['name'].' : '.$user['description'].'<br>';
7     }
8 } catch (PDOException $e) {
9     echo 'Impossible de récupérer la liste des jeux';
10 }

```

p. 9 Solution n°2

```

1 SELECT * FROM games WHERE name LIKE :search

```

p. 9 Solution n°3

```

1 SELECT * FROM games WHERE description LIKE ?

```

p. 15 Solution n°4

```

1 <?php
2
3 try {
4     $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
5     $statement = $pdo->prepare('DELETE FROM games WHERE id = :id');
6     $statement->bindValue(':id', $_GET['id'], PDO::PARAM_INT);
7     if ($statement->execute()) {
8         echo 'La suppression a bien été effectuée';
9     } else {
10         $errorInfo = $statement->errorInfo();
11         echo $errorInfo[2];
12     }
13 } catch (PDOException $e) {
14     echo 'Une erreur s\'est produite lors de la communication avec la base';
15 }

```

p. 15 Solution n°5

```

1 <?php
2
3 try {
4     $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
5     // Erreur de requête volontaire pour tester l'erreur
6     $statement = $pdo->prepare('DELETE FROM games WHERE id = :id');
7     $statement->bindValue(':id', $_GET['id'], PDO::PARAM_INT);
8     if ($statement->execute()) {
9         echo 'La suppression a bien été effectuée';
10     } else {

```

```

11     $insertStatement = $pdo->prepare('INSERT INTO errors(state, driverError, message)
VALUES (?, ?, ?)');
12     $errorInfo = $statement->errorInfo();
13     $insertStatement->execute($errorInfo);
14     echo $errorInfo[2];
15 }
16 } catch (PDOException $e) {
17     echo 'Une erreur s\'est produite lors de la communication avec la base';
18 }

```

p. 20 Solution n°6

```

1 <?php
2
3 // Game.php
4 class Game
5 {
6     private int $id;
7     private string $name;
8     private string $description;
9
10    public function getFullDescription()
11    {
12        return $this->name . ' - ' . $this->description . '<br>';
13    }
14 }
15
16 // index.php
17
18 try {
19     require_once 'Game.php';
20     $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
21     $statement = $pdo->prepare('SELECT * FROM games WHERE name LIKE :search');
22     $statement->bindValue(':search', $_GET['search'] . '%', PDO::PARAM_STR);
23     $statement->setFetchMode(PDO::FETCH_CLASS, 'Game');
24     if ($statement->execute()) {
25         while ($user = $statement->fetch()) {
26             echo $user->getFullDescription();
27         }
28     } else {
29         $errorInfo = $statement->errorInfo();
30         echo $errorInfo[2];
31     }
32 } catch (PDOException $e) {
33     echo 'Une erreur s\'est produite lors de la communication avec la base';
34 }

```

p. 22 Solution n°7

```

1 <?php
2
3 // Game.php
4 class Game
5 {
6     private int $id;
7     private string $name;

```

```


8     private string $description;
9
10    public function getFullDescription()
11    {
12        return $this->name . ' - ' . $this->description;
13    }
14 }
15
16 // index.php
17 try {
18     require_once 'Game.php';
19     $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
20     $statement = $pdo->prepare('SELECT * FROM games LIMIT :start, 5');
21     $statement->bindValue('start', 5 * ($_GET['page'] - 1), PDO::PARAM_INT);
22     $statement->setFetchMode(PDO::FETCH_CLASS, 'Game');
23     if ($statement->execute()) {
24         while ($user = $statement->fetch()) {
25             echo $user->getFullDescription();
26         }
27     } else {
28         $errorInfo = $statement->errorInfo();
29         echo $errorInfo[2];
30     }
31 } catch (PDOException $e) {
32     echo 'Une erreur s\'est produite lors de la communication avec la base';
33 }

```

Exercice p. 23 Solution n°8


Exercice

À quoi sert la méthode `query` ?

- ☐ Se connecter à la base de données
- ☒ Lancer une requête et récupérer le résultat
- ☐ Lancer une requête et récupérer le nombre de lignes affectées
-  La méthode `query` permet de lancer une requête et de récupérer le résultat : il faut donc l'utiliser avec une requête `SELECT`.

Exercice


Lorsque l'on récupère le résultat d'une requête `SELECT` en PHP avec le mode `PDO::FETCH_ASSOC`, comment est déterminé le nom de chaque clé ?

- ☐ C'est la position de chaque colonne dans la requête
- ☐ C'est le nom de chaque colonne des tables concernées par la requête
- ☒ C'est le nom de chaque colonne de la table virtuelle retournée par la requête
-  Le nom de chaque clé est bien le nom de la table virtuelle retournée par la requête, ce qui nous permet de leur donner des alias.

Exercice

Soit la table **garages**, ayant pour champs `id`, `name` et `address` et la table **cars**, ayant pour champs `id`, `garageId`, `name` et `brand`. Combien de colonnes est-ce que le résultat de la requête `SELECT * FROM garages JOIN cars ON garages.id = cars.garageId` possédera-t-il en utilisant le mode `PDO::FETCH_ASSOC` ?


- ☐ 7
- ☐ 6
- ☒ 5
- ☐ 4
- ☐ 3

 Un tableau associatif en PHP ne peut pas avoir plusieurs fois le même nom de colonne. Le tableau représentant le résultat de cette requête possédera donc 5 champs : `id` (écrasé), `name` (écrasé), `address`, `garageId` et `brand`.

Exercice

Pour les mêmes tables et la même requête que la question précédente, combien de champs possédera le tableau de résultats récupérés en utilisant le mode `PDO::FETCH_NUM` ?


- ☒ 7
- ☐ 6
- ☐ 5
- ☐ 4
- ☐ 3

 Le mode `PDO::FETCH_NUM` ne se base pas sur le nom des colonnes, mais sur leur position : il n'y a donc aucun risque d'écrasement, et on retrouve bien toutes les valeurs. Il y a donc bien 7 champs.

Exercice


Parmi ces requêtes préparées, lesquelles sont valides ?

- ☒ `SELECT * FROM cars WHERE brand LIKE :search`
- ☐ `SELECT * FROM cars WHERE brand LIKE :search%`
- ☒ `SELECT * FROM cars WHERE brand LIKE ?`
- ☐ `SELECT * FROM :table WHERE brand LIKE :search`
- ☐ `SELECT * FROM cars WHERE $field LIKE :search`

 L'avantage des requêtes préparées est de permettre au SGBD d'optimiser leur exécution en connaissant à l'avance les tables et les champs qui les composent. Il n'est donc pas possible de paramétrer ces éléments de la requête : seules des valeurs unitaires peuvent être remplacées par des marqueurs.

Exercice

Qu'est-ce qu'un objet `PDOStatement` ?


- ☐ La représentation d'une connexion à la base de données
- ☒ La représentation d'une requête
- ☐ La représentation d'une ligne de données
-  Un `PDOStatement` représente une requête et permet de réaliser toutes les actions dessus : renseigner les valeurs de ses marqueurs, l'exécuter, récupérer ses résultats...

Exercice

Le code ci-dessous est valide.


```

1 <?php
2
3 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
4 $statement = $pdo->prepare('SELECT * FROM cars WHERE brand LIKE :search');
5 $statement->bindParam(':search', 'audi%', PDO::PARAM_STR);
6 if ($statement->execute()) {
7     while ($car = $statement->fetch(PDO::FETCH_ASSOC)) {
8         echo $car['name'];
9     }
10 }
```

- ☐ Vrai
- ☒ Faux
-  Le code ci-dessus n'est pas valide. En effet, il n'est pas possible d'utiliser `bindParam` avec une valeur : il faut obligatoirement lui donner une variable, puisque la liaison se fait par référence. Dans ce cas, il serait préférable d'utiliser `bindValue`.


Exercice

À quoi sert le mode de récupération `PDO::FETCH_OBJ` ?

- ☒ Récupérer les données sous forme d'instance de `stdClass`, la classe standard de PHP
- ☐ Récupérer les données sous forme d'instance d'objet de notre application : il faut alors lui donner le nom de la classe à instancier
- ☐ Récupérer les données sous forme d'un mélange entre le tableau associatif et indexé
-  `PDO::FETCH_OBJ` permet de récupérer les résultats sous forme d'objets `stdClass`. Il est préférable d'utiliser `PDO::FETCH_CLASS`, qui permet de renseigner le nom d'une classe de notre application.

Exercice


Combien de résultats va retourner la requête `SELECT * FROM cars LIMIT 5, 10` ?

- ☐ Au moins 5
- ☐ Au plus 5
- ☐ Au moins 10
- ☒ Au plus 10
-  Lorsque deux nombres sont renseignés dans la clause `LIMIT`, alors le second détermine le nombre maximal de résultats.

Exercice

Comment faire pour récupérer les résultats de la page X de mon site, sachant qu'il y a 12 voitures par page ?

- ☐ SELECT * FROM cars LIMIT X, 12
- ☐ SELECT * FROM cars LIMIT X * 12, 12
- ☒ SELECT * FROM cars LIMIT (X - 1) * 12, 12
- ☐ SELECT * FROM cars LIMIT (X + 1) * 12, 12

 Pour récupérer la page 1, il faut récupérer les résultats 0 à 11. Pour la page 2, il faut récupérer les résultats 12 à 23, etc. Donc, pour la page X, nous allons devoir récupérer les résultats : $(X - 1) * 12$ à $X * 12 - 1$.

p. 25 Solution n°9

```
1 CREATE TABLE messages(
2     id INTEGER(11) PRIMARY KEY AUTO_INCREMENT,
3     content VARCHAR(1000),
4     date DATETIME DEFAULT CURRENT_TIMESTAMP
5 )
```

Fichier Message.php :

```
1 <?php
2
3 class Message
4 {
5     private int $id;
6     private string $content;
7     private string $date;
8
9     public function getContent(): string
10    {
11        return $this->content;
12    }
13
14    public function getDate(): string
15    {
16        return $this->date;
17    }
18 }
```

p. 25 Solution n°10

Fichier view.php :

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Mon fil d'actualités</title>
6 </head>
7 <body>
8     <form method="post">
9         <textarea name="content"></textarea>
10        <br>
11        <button type="submit">Créer un message</button>
```



```

12     </form>
13 </body>
14 </html>

```

Fichier index.php:

```

1 <?php
2
3 // Le contrôleur s'occupe d'inclure la vue.
4 include 'view.php';

```

p. 26 Solution n°11

Fichier MessageManager.php:

```

1 <?php
2
3 class MessageManager
4 {
5     private PDO $pdo;
6
7     public function __construct(PDO $pdo)
8     {
9         // Injection de dépendances de l'objet PDO
10         $this->pdo = $pdo;
11     }
12
13     public function addMessage(string $message): bool
14     {
15         // L'identifiant et la date du message sont calculés automatiquement par le SGBD.
16         // Seul le contenu doit être inséré.
17         $statement = $this->pdo->prepare('INSERT INTO messages (content) VALUES (:content)');
18         $statement->bindValue(':content', $message, PDO::PARAM_STR);
19
20         return $statement->execute();
21     }
22 }

```

Fichier index.php:

```

1 <?php
2
3 require_once 'MessageManager.php';
4
5 // On crée une instance de PDO et on l'injecte dans notre manager
6 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
7 $manager = new MessageManager($pdo);
8
9 // On stocke l'affichage du message de succès dans une variable $hasSuccessAlert, qui sera
10 // utilisée dans la vue
11 $hasSuccessAlert = false;
12 if (!empty($_POST['content'])) {
13     // Si un contenu a été posté, alors on ajoute le message.
14     $hasSuccessAlert = $manager->addMessage($_POST['content']);
15 }
16 include 'view.php';

```

Fichier `view.php`:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Mon fil d'actualités</title>
6 </head>
7 <body>
8 <!-- On regarde la valeur de $hasSuccessAlert et affiche le message en conséquence -->
9 <?php if ($hasSuccessAlert) { ?>
10     <b>Votre message a bien été ajouté !</b>
11 <?php } ?>
12     <form method="post">
13         <textarea name="content"></textarea>
14         <br>
15         <button type="submit">Créer un message</button>
16     </form>
17 </body>
18 </html>

```

p. 26 Solution n°12

Fichier `Message.php`:

```

1 <?php
2
3 class Message
4 {
5     private int $id;
6     private string $content;
7     private string $date;
8
9     public function getContent(): string
10    {
11        return $this->content;
12    }
13
14    public function getDate(): string
15    {
16        return $this->date;
17    }
18 }

```

Fichier `MessageManager.php`:

```

1 <?php
2
3 class MessageManager
4 {
5     private PDO $pdo;
6
7     public function __construct(PDO $pdo)
8     {
9         $this->pdo = $pdo;
10    }
11
12    public function addMessage(string $message): bool
13    {

```

```

14         $statement = $this->pdo->prepare('INSERT INTO messages (content) VALUES (:content)');
15         $statement->bindValue(':content', $message, PDO::PARAM_STR);
16
17         return $statement->execute();
18     }
19
20     public function getMessages(int $page): array
21     {
22         require_once 'Message.php';
23         $statement = $this->pdo->prepare('SELECT * FROM messages LIMIT :start, 10');
24         // On récupère les messages sous forme d'objets Message
25         $statement->setFetchMode(PDO::FETCH_CLASS, 'Message');
26         // La position de départ du LIMIT dépend de la page. Attention au type de données,
27         // qui doit être en PARAM_INT
28         $statement->bindValue(':start', 10 * ($page - 1), PDO::PARAM_INT);
29         $statement->execute();
30
31         return $statement->fetchAll();
32     }
33 }

```

Fichier index.php

```

1 <?php
2
3 require_once 'MessageManager.php';
4
5 $pdo = new PDO('mysql:host=localhost;dbname=intro_pdo', 'root', '');
6 $page = $_GET['page'] ?? 1;
7 $manager = new MessageManager($pdo);
8 $hasSuccessAlert = false;
9 if (!empty($_POST['content'])) {
10     $hasSuccessAlert = $manager->addMessage($_POST['content']);
11 }
12
13 //On stocke les messages dans une variable $messages, qui sera utilisée dans la vue.
14 $messages = $manager->getMessages($page);
15 include 'view.php';

```

Fichier messageView.php:

```

1 <!-- La vue met en forme un objet de type Message contenu dans une variable $message -->
2 <div>
3     <p><?php echo $message->getContent(); ?></p>
4     <i>Le <?php echo $message->getDate(); ?></i>
5 </div>

```

Fichier view.php:

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4     <meta charset="UTF-8">
5     <title>Mon fil d'actualités</title>
6 </head>
7 <body>
8 <?php if ($hasSuccessAlert) { ?>
9     <b>Votre message a bien été ajouté !</b>
10 <?php } ?>
11     <form method="post">
12         <textarea name="content"></textarea>
13         <br>

```

```
14         <button type="submit">Créer un message</button>
15     </form>
16
17 <?php
18     // Pour tous les messages de $messages, on appelle la vue messageView en y injectant le
    message à afficher.
19     foreach ($messages as $message) {
20         // Attention a ne pas utilise include_once ici : le but est d'inclure la vue autant
    de fois qu'il y a de messages
21         include 'messageView.php';
22     }
23 ?>
24 </body>
25 </html>
```

Vous avez maintenant votre propre Twitter entre vos mains !

N'hésitez pas à essayer d'améliorer ce code par vous-même : même si cette application est fonctionnelle, il reste malgré tout beaucoup de choses à y ajouter. Vous pourriez ajouter une liste de pages pour naviguer sans avoir à modifier l'URL, augmenter la sécurité de votre application en vous assurant que le numéro de page passé en paramètre est bien un nombre entier, améliorer le style de la page, ajouter un champ permettant de rechercher un message selon son contenu... Vous avez les connaissances nécessaires pour réaliser tout cela : la seule limite est celle de votre imagination !