

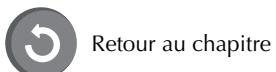
Utiliser PDO et les requêtes préparées

Sommaire

I. Les risques de l'injection SQL	3
II. PDO et les requêtes préparées (prepared statements)	3
A. Avantages.....	4
B. Principe général de mise en œuvre.....	4
C. Instanciation puis utilisation de PDO	4
D. Une requête préparée : PDO avec variables	6
E. Cas d'une requête avec deux variables ou plus.....	7
III. Gestion des erreurs	8
A. Utilisation de try, throw et catch, les exceptions.....	8
B. Gestion des erreurs MySQL.....	11

Crédits des illustrations:
© Fotolia, DR.

Les repères de lecture



Retour au chapitre



Définition



Objectif(s)



Espace Élèves



Vidéo /
Audio



Point important /
À retenir



Remarque(s)



Pour aller
plus loin



Normes et lois



Quiz

Les requêtes préparées (prepared statements en anglais) sont un moyen sécurisé d'intégrer des requêtes SQL dans un code PHP.

Elles sont devenues un standard de mise en œuvre incontournable dans tout projet PHP professionnel.



Fig. 1

I. Les risques de l'injection SQL



PHP représente une interface entre l'utilisateur et la base en permettant l'utilisation de requêtes SQL, or le problème majeur est que les requêtes SQL peuvent être manipulées par l'injection SQL. Une littérature abondante existe sur le sujet et on pourra entre autres se référer à la présentation des risques du manuel : http://php.net/manual/fr/security_database.sql-injection.php

Rappelons quelles sont les problématiques de sécurité PHP/MySQL.

Le contenu d'un site représente la valeur stratégique du site, qu'il s'agisse d'articles pour un site de presse ou des listings clients pour un site e-commerce. La très grande majorité des sites utilise une base de données (MySQL ou autre) pour stocker leur contenu. Il est donc clair que la base de données est l'enjeu essentiel de la sécurité. Sont visés ici les utilisateurs malveillants professionnels, ou malgré eux.

Pour éviter que les utilisateurs malveillants ajoutent du contenu non souhaité dans la base ou s'attribuent les droits d'administrateur, le meilleur moyen est d'utiliser PDO et les requêtes préparées.

II. PDO et les requêtes préparées (prepared statements)

PDO (PHP Data Objects) est un système qui va gérer la relation entre PHP et un serveur de base de données dans un principe d'intermédiation. Il s'agit donc d'une interface entre PHP et MySQL. PDO est installé sur la majorité des environnements PHP depuis la version PHP 5. C'est une extension de PHP qui devient le standard de connexion à la base de données.

Il est nécessaire de bien étudier la syntaxe spécifique des requêtes préparées.

A. Avantages



Il faut tout de même vérifier en amont que les informations transmises par l'utilisateur sont bien du type attendu. Il est également impératif de se protéger lors de l'enregistrement des insertions de code HTML (avec `htmlspecialchars`) ainsi qu'à la sortie (avec `htmlentities`).

- Les requêtes préparées sont exécutées plus rapidement et consomment moins de ressources que les requêtes SQL classiques;
- Les requêtes préparées empêchent totalement l'injection SQL sans qu'il soit nécessaire d'utiliser des fonctions PHP de nettoyage des valeurs, comme avec `mysql_real_escape_string` par exemple.

B. Principe général de mise en œuvre

PDO agissant comme une couche intermédiaire, il est nécessaire de « connecter PHP à PDO », à l'instar finalement de la connexion de PHP à MySQL, lorsque nous affectons à une variable `$mysqli` les informations de connexion à la base de données (`$mysqli = new mysqli('localhost', 'root', '', 'projet_villes')`).

Puisque PDO est une interface entre PHP et MySQL, il faut commencer par déclarer que nous allons utiliser PDO.

Ensuite, **puisque PDO est construit en PHP sous la forme d'une classe**, pour utiliser PDO dans un projet, nous devons donc commencer par instancier PDO ! PDO s'utilise donc comme une classe qui retourne des paramètres de connexion. Après avoir vu comment déclarer PDO, nous pourrons ensuite l'utiliser librement dans nos projets et créer nos requêtes.

En résumé, le principe général de mise en œuvre est le suivant:

- **instanciation de PDO;**
- **usage dans les classes** (utilisation des requêtes préparées).

C. Instanciation puis utilisation de PDO

<http://fr2.php.net/manual/fr/pdo.connections.php>

Plusieurs versions sont possibles en fonction des détails de gestion souhaités (erreur, etc.).

1. Instanciation

```
$dbh = new PDO('mysql:host=localhost;dbname=dbemail', 'root', '');
```

Fig.2

Comme on le voit avec `new`, une instance de PDO est créée contenant les informations classiques de connexion au serveur de base de données : l'adresse du serveur (ici `localhost` avec Wamp), le nom de la base de données (ici `dbemail`), le *login* utilisateur MySQL (ici `root`) et le mot de passe MySQL (qui ici est vide).

Le résultat de cette instance est affecté à une variable `$dbh` (qui est le nom de variable utilisé par le manuel PHP et que nous garderons, `dbh` pour *database handler, to handle = gérer*).

2. Usage

La variable `$dbh` contient l'objet PDO qui offre un ensemble de méthodes. La première méthode que nous utiliserons est la méthode *prepare* qui constitue la première étape de l'échange entre PHP, PDO et MySQL.

La méthode *prepare* aura pour argument la requête SQL souhaitée :

```
$stmt = $dbh->prepare('SELECT id, email FROM email');
```

Fig.3

Le résultat est affecté à la variable `$stmt` (pour *statement*).

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

Fig.4

Puis, la méthode PDO *execute* demande à PDO d'exécuter la requête auprès de la base de données (à la manière de *mysqli_query* lorsque nous utilisons l'ancienne méthode).

Il ne reste plus qu'à récupérer les informations :

```
while ($row = $stmt->fetch())  
{  
    print_r($row);  
}
```

Fig.5

Le code complet de la requête avec PDO est donc :

```
$dbh = new PDO('mysql:host=localhost;dbname=dbemail', 'root', '';  
$stmt = $dbh->prepare('SELECT id, email FROM email');  
$stmt->execute();  
while ($row = $stmt->fetch()) {  
    print_r($row);  
}
```

Fig.6

Le résultat retourné contient les informations pour l’ensemble des trois emails enregistrés dans la table.

```
Array
(
    [0] => Array
        (
            [id] => 1
            [0] => 1
            [email] => mail01@anyhost
            [1] => mail01@anyhost
        )
    [1] => Array
        (
            [id] => 2
            [0] => 2
            [email] => mail02@anyhost
            [1] => mail02@anyhost
        )
    [2] => Array
        (
            [id] => 3
            [0] => 3
            [email] => mail03@anyhost
            [1] => mail03@anyhost
        )
)
```

Fig.7

D. Une requête préparée: PDO avec variables

Si nous souhaitons afficher uniquement l’adresse email *mail01@anyhost*:

```
# Instanciation
$dbh = new PDO('mysql:host=localhost;dbname=dbemail', 'root', '');
# Code
$email = 'mail01@anyhost' ; // adresse email recherchée
$stmt = $dbh->prepare('SELECT id, email FROM email WHERE email
=:email');
$stmt->bindParam('email', $email);
$stmt->execute();
while ($row = $stmt->fetch()) {
    print_r($row);
}
```

Fig.8

Le résultat retourné retourne bien les informations pour la valeur spécifiée.

```
Array
(
    [0] => Array
        (
            [id] => 1
            [0] => 1
            [email] => mail01@anyhost
            [1] => mail01@anyhost
        )
)
```

Fig.9



Le principe du bind

Le principe du bind est que les noms des champs et des variables sont considérés comme des paramètres qui seront bindés – du verbe anglais *to bind*: lier, attacher – et que c'est PDO qui finalisera la construction de la requête finale avec d'un côté le **SELECT** et de l'autre côté la variable bindée.

Nous constatons que la syntaxe est particulière:

- appel d'une méthode *prepare*: destinée à fournir à PDO la requête SQL;
- utilisation de double points (:) devant *email* dans la requête SQL (... WHERE email =:*email*) : nécessaire pour le *bind* (voir ci-dessous);
- appel d'une méthode *bindParam*: création du *bind* (c'est-à-dire la liaison, voir ci-dessous);
- appel d'une méthode *execute*: demande à PDO d'exécuter la requête vers la base de données;
- appel d'une méthode *fetch*: demande à PDO de restituer les résultats fournis par la base de données.

The diagram shows a snippet of PHP code with annotations. A red circle labeled '1' points to the line '\$stmt->bindParam('email', \$email);'. A red circle labeled '2' points to the line 'WHERE email =:email' in the SQL query. The code is as follows:

```
$email = 'mail01@anyhost';
$stmt = $dbh->prepare("SELECT id, email FROM emails WHERE email =:email");
$stmt->bindParam('email', $email);
```

Fig. 10

Le *bind* en deux temps:

- affectation de la variable *\$email* au paramètre PDO '*email*' ;
- utilisation du paramètre '*email*' précédé du double point dans la requête, cette requête étant elle-même un argument de la méthode *prepare*.

E. Cas d'une requête avec deux variables ou plus

```
# Instanciation
$dbh = new PDO('mysql:host=localhost;dbname=dbemail', 'root', '');

# Code
// exemple de variables
$email = 'mail01@anyhost'; // adresse email recherchée
$name = 'Martin'; // nom recherché

$stmt = $dbh->prepare('SELECT id, email FROM email
    WHERE email =:email
    AND
    name =:name');
$stmt->bindParam('email', $email);
$stmt->bindParam('name', $name);
$stmt->execute();
while ($row = $stmt->fetch()) {
    print_r($row);
}
```

Fig. 11

Comme on le voit, le *bind* est utilisé pour chaque variable que nous souhaitons attacher, puis les paramètres créés sont intégrés classiquement dans la requête SQL.

Il est important (et agréable) de noter qu'il n'y a pas de *quote* utilisé dans la construction de la requête.

L'ensemble des requêtes SQL peut être utilisé. Nous ne passerons pas en revue toutes les requêtes possibles mais voici deux exemples :

1. INSERT INTO

Ici une variante : la requête est préalablement affectée à une variable pour simplifier le code.

```
$sql = "INSERT INTO email (email) VALUES (:email)";
$stmt = $dbh->prepare($sql);
$stmt->bindParam(':email', $email);
```

Fig. 12

2. UPDATE

```
$sql = "UPDATE email SET email =:email WHERE id=:id";
$stmt = $dbh->prepare($sql);
$stmt->bindParam(':email', $email)
$stmt->bindParam(':id', $id);
```

Fig. 13

III. Gestion des erreurs

A. Utilisation de try, throw et catch, les exceptions



Une **exception** n'est pas un bug mais une **mise en œuvre décidée par le développeur** qui permet de gérer les différents cas résultant de l'exécution d'un script, comme par exemple un mot de passe non valide, une saisie non conforme, etc.

Pour gérer les erreurs de connexion au serveur de base de données, il est pratique d'utiliser les **exceptions** avec les fonctions natives *try*, *throw* et *catch*.

Try (essayer), *throw* (lancer) et *catch* (intercepter, récupérer au vol) permettent respectivement d'exécuter le code dans le bloc *try*, de lancer une **exception** avec *throw* et de récupérer l'exception dans le bloc *catch*.

Exemple

Try et *catch* fonctionnent comme une structure conditionnelle.

```
try
{
    throw new Exception('Un message !');

}
catch(Exception $e) // intercepte l'exception lancée par throw
{
    echo $e->getMessage(); // affiche : un message !
}
```

Fig. 14

Afin de regarder la construction et la syntaxe, nous nous limitons à un simple *throw* dans le bloc *try* (qui pourrait contenir du code supplémentaire) et au bloc *catch*.

Nous constatons l'usage du mot-clé *new* qui indique que nous allons instancier la classe *Exception*. *Exception* est en effet la classe de base pour toutes les exceptions. Page du manuel PHP : <http://www.php.net/manual/fr/class.exception.php>

L'argument contient ensuite notre message dont le contenu est « Un message ! » et qui sera repris pour être affiché ensuite dans le bloc *catch*.

Notons que *catch* attrape au vol l'exception lancée par *throw* et que *throw* interromps l'exécution du code. Dans l'argument qui débute le bloc *catch*, la syntaxe (*Exception \$e*) signifie que c'est à cet instant que la classe *Exception* est alors instanciée par l'objet *\$e* (objet qui est noté *\$e* par convention).

Une fois cette instance réalisée, il suffit d'utiliser les méthodes de la classe, ici *getMessage*.

Un *var_dump* sur *\$e* nous donne les informations suivantes :

```
var_dump ($e);
object(Exception) [1]
protected 'message' => string 'Un message !' (length=12)
private 'string' => string '' (length=0)
protected 'code' => int 0
protected 'file' => string 'C:\wamp\www\emweb_livret4\demo\
exception_sql.php' (length=48)
protected 'line' => int 30
private 'trace' =>
array
empty
private 'previous' => null
public 'xdebug_message' => string '<tr>... (length=854)
```

Fig. 15

\$e est donc bien un objet, et nous pourrions donc utiliser un *get_class_methods* afin de vérifier quelles méthodes sont disponibles pour accéder aux propriétés (car elles sont en mode *protected* ou *private* et donc non accessibles directement depuis l'extérieur de la classe).

```

array
0 => string '__construct' (length=11)
1 => string 'getMessage' (length=10)
2 => string 'getCode' (length=7)
3 => string 'getFile' (length=7)
4 => string 'getLine' (length=7)
5 => string 'getTrace' (length=8)
6 => string 'getPrevious' (length=11)
7 => string 'getTraceAsString' (length=16)
8 => string '__toString' (length=10)

```

Fig. 16

Le manuel le spécifie pour chaque méthode mais nous savons que les méthodes commençant par `get` sont généralement dédiées à l'obtention des propriétés, par exemple afficher le message avec `getMessage`, le code (d'erreur SQL) avec `getCode`, la ligne avec `getLine`, etc.

```

echo 'Message d\'erreur : '. $e->getMessage();
echo 'Code erreur SQL : '. $e->getCode();
echo 'Fichier concerné : '. $e->getFile();
echo 'Numéro de ligne : '. $e->getLine();

```

Fig. 17

1. Exemple concret

Nous pouvons utiliser les exceptions pour gérer une vérification.

```

try
{
    $var = 'foo';
    $user = 2;

    if($var != $user)
    {

        throw new Exception('la valeur ' . $user . ' est incorrecte.');
    }
    // ne sera pas affiché car throw interromps l'exécution du code
    echo 'affiche ce texte';
}
catch(Exception $e)
{

    echo 'Une erreur est survenue : ';
    echo $e->getMessage();
}

```

Fig. 18

À l'instar de la structure, `if et else`, `try et catch` peuvent également être imbriqués. À ce propos, puisque nous pourrions naturellement gérer ce cas avec `if et else`, pourquoi le faire avec `try et catch`?

Pour répondre à cette question, il faut se demander si le bon fonctionnement de ce qui est mis en œuvre est attendu, comme par exemple pour une connexion à un serveur de base de données. Si la connexion ne s'établit pas, il s'agit bien d'une exception (au sens courant du terme). Dans ce cas, *try/catch* sera utilisée à bon escient.

Concernant la gestion de cas tels qu'une saisie d'un *login* utilisateur incorrect ou bien la vérification de valeur d'une variable, la structure *if/else* sera préférable, car la valeur de la variable pourra éventuellement ne pas correspondre et qu'il n'y a rien d'exceptionnel à cela (donc pas d'exception).

2. Application des exceptions à la connexion PDO

Une première mise en œuvre permettra de gérer une erreur de connexion au serveur de base de données (ici sans utiliser le *throw*) :

```
try
{
    $dbh = new PDO('mysql:host=localhost;dbname=dbemail', 'root', '');
}
catch(Exception $e)
{
    echo 'Message erreur SQL : '.$e->getMessage().'  
>';
    exit;
}
```

Fig.19

Si nous indiquons un nom erroné de la base de données :

```
Message Erreur : SQLSTATE[42000] [1049] Unknown database 'dbemails'
```

Fig.20

Notons que la deuxième valeur entre crochets [1049] est le code erreur de MySQL.

B. Gestion des erreurs MySQL

La gestion des erreurs de connexion à la base de données étant effectuée, nous pouvons ensuite nous occuper de gérer les éventuelles erreurs de MySQL.

Par exemple, notre table contenant des emails, il peut être pertinent d'envisager de ne pas insérer un email déjà enregistré auparavant afin d'éliminer les doublons.

Il est envisageable de traiter ce besoin dans le script PHP en effectuant une requête préalable à l'enregistrement et de ne pas effectuer celui-ci si le résultat de la requête n'est pas nul.

Mais un moyen encore plus simple est d'utiliser MySQL en ajoutant un index unique sur la colonne email.

Nous avions déjà vu que la clef primaire (*Primary Key*) était un index de la table mais d'autres index peuvent être ajoutés, et notamment l'index **unique**. L'ajout de l'index unique sur une colonne fera que toutes les valeurs de cette colonne seront uniques et donc aucun doublon ne pourra s'y trouver.

La requête MySQL est la suivante:

```
ALTER TABLE email ADD UNIQUE (email)
```

Fig.21

```
ERROR 1062 (23000): Duplicate entry 'email@anyhost.com' for key 'email'
```

Fig.22

C'est d'ailleurs justement ce retour de MySQL que nous allons exploiter dans le cas présent.

L'ajout d'index étant effectué sur la table *email*, notre script d'insertion d'adresses email peut être produit:

```
try
{
    $dbh = new PDO('mysql:host=localhost;dbname=dbemail', 'root', '');
}
catch(Exception $e)
{

echo 'Message erreur SQL : '.$e->getMessage().'  

```

Fig.23

En l'état, aucun message n'est retourné et nous ne savons pas si l'insertion a réussi. Nous allons utiliser PDO pour obtenir des informations.

La variable *\$stmt*, à laquelle nous affectons le résultat de la méthode *prepare* de l'objet *\$dbh*, nous permet de simplifier l'écriture pour utiliser les méthodes *bindParam* et *execute*.

Il s'agit donc également d'un objet, et un `var_dump` nous retourne les informations suivantes :

```
object(PDOStatement) [2]
  public 'queryString' => string 'INSERT INTO email (email) VALUES (:email)' (length=41)
```

Fig.24

Si c'est un objet nous pouvons avec la fonction PHP native `get_class_methods` afficher les méthodes de sa classe.

```
array
  0 => string 'execute' (length=7)
  1 => string 'fetch' (length=5)
  2 => string 'bindParam' (length=9)
  3 => string 'bindColumn' (length=10)
  4 => string 'bindValue' (length=9)
  5 => string 'rowCount' (length=8)
  6 => string 'fetchColumn' (length=11)
  7 => string 'fetchAll' (length=8)
  8 => string 'fetchObject' (length=11)
  9 => string 'errorCode' (length=9)
  10 => string 'errorInfo' (length=9)
  11 => string 'setAttribute' (length=12)
  12 => string 'getAttribute' (length=12)
  13 => string 'columnCount' (length=11)
  14 => string 'getColumnMeta' (length=13)
  15 => string 'setFetchMode' (length=12)
  16 => string 'nextRowset' (length=10)
  17 => string 'closeCursor' (length=11)
  18 => string 'debugDumpParams' (length=15)
  19 => string '__wakeup' (length=8)
  20 => string '__sleep' (length=7)
```

Fig.25

Nous constatons que certaines de ces méthodes sont déjà utilisées (`execute`, `bindParam`, `fetch`). Une méthode semble pourvoir nous fournir les informations sur le succès ou l'erreur de notre requête : `errorInfo`.

Nous pouvons donc tester cette méthode :

```
$errors = $stmt->errorInfo();
var_dump ($errors);
```

Fig.26

Ce qui nous permet d'obtenir le tableau suivant :

```
array
  0 => string '23000' (length=5)
  1 => int 1062
  2 => string 'Duplicate entry \'email@anyhost.com\' for key \'email\''
  (length=51)
```

Fig.27

Nous obtenons donc le script définitif suivant:

```
$email = 'une valeur';
$sql = "INSERT INTO email (email) VALUES (:email)";
$stmt = $dbh->prepare($sql);
$stmt->bindParam(':email', $email);
$stmt->execute();

// gestion des erreurs SQL
$errors = $stmt->errorInfo();
if ($errors[0] != '00000')
{

    echo 'Erreur SQL ' . $errors[2];
}
else
{

    echo 'L\'enregistrement de l\'email est bien effectué';
}
```

Fig.28