Einführung in PDO

PDO(PHP Data Object) ist der moderne Weg mit PHP5 auf die Datenbank zuzugreifen. Es bietet eine Abstraktionsschicht für den Datenzugriff. Unabhängig vom verwendeten DBMS können die selben Funktionen verwenden.
PDO räumt außerdem mit vielen Sicherheitsängsten auf, die man von anderen Datenbank-Schnittstellen kennt. Die vielen Wege die teilweise zum selben Ergebnis führen schrecken viele Programmierer ab, daher soll dieses Tutorial einen konsistenten Weg liefern.
Konfiguration
Eine gängige Art die Konfigurationvariablen zu hinterlegen ist eine finale Klasse mit Konstanten. Sie benötigt relativ wenig Prozess Overhead.

Quellcode

```
1. final class Configuration {
2. const DB_HOST='localhost';
3. const DB_DATABASE='datebase';
4. const DB_PORT=3306;
5. const DB_USER='database_user';
6. const DB_PASSWORD='database_password';
7. }
```

== Verbindung herstellen ==
Einleitend wurde erläutert, dass PDO eine Abstraktionsschicht für den Datenbankzugriff bereitstellt. Durch Abstraktion folgert Einschränkung - nämlich auf den Nenner des schwächsten unterstützen DBMS. Aus mehreren Gründen macht es Sinn den Datenbankzugriff weiter zu kapseln.
Wir erstellen uns daher eine neue Klasse "MyDB" die als Singleton implementiert wird, somit wird verhindert, dass in der selben PHP Anwendung mehrmals die Verbindung hergestellt wird.
Außerdem erstellen wir eine persistente Verbindung - das bietet sich bei den meisten Webanwendungen an.

Quellcode

```
1. class MyDB {
2. private static $db;
3. static public function getInstance() {
4. if(!self::$db) {
5. self::$db = new PDO(
6. 'mysql:host='.Configuration::DB_HOST.';dbname='.Configuration::DB_DATABASE.';port='.Configuration::DB_PORT,
7. Configuration::DB_USER,
8. Configuration::DB_PASSWORD,
9. array(
10. PDO::ATTR_PERSISTENT => true,
11. PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION ,
12. PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC
13. )
14. );
15. }
16. return self::$db;
17. }
18. }
```

Alles anzeigen

== Datenbankänderungen ==
Die einfachste Form von Datenbankzugriffen stellt das Ändern kleiner Daten dar. Parameter die man per Formular übergibt oder über die URL erhält, werden als Array-Parameter durch das execute() Statement gebunden.
Außerdem fällt auf, dass der SQL-String durch das prepare Statement vorbereitet wird. Das ist ein Performance Faktor, sollte man das selbe "Prepared Statement" mehrmals mit unterschiedlichen Parameter ausführen wollen (was ein extrem

seltener Fall ist).

Quellcode

```
$pdoparams = array(
':userid' => $_REQUEST['userid'],
':website' => 'http://www.easy-coding.de'
);
$sql = "UPDATE user
SET website = :website
WHERE userid = :userid
LIMIT 1";
$stmt = MyDB::getInstance()->prepare($sql);
$stmt->execute($pdoparams);
```

== Neue Datenbankeintrage: ID des Datensatzes ==
Werden auto_increment (MySQL) oder SERIAL (postgreSQL) Felder genutzt, kümmert sich das DBMS bei neuen Einträgen um die neuen Primärschlüssel. Sie werden mit der Funktion lastInsertID abgefragt.

Quellcode

```
$pdoparams = array(
':website' => 'http://www.easy-coding.de'
);
$sql = "INSERT INTO user
(website)
VALUES (:website)";
$stmt = MyDB::getInstance()->prepare($sql);
$stmt->execute($pdoparams);
$userID = MyDB::getInstance()->lastInsertId();
echo $userID;
```

== Datenbankabfragen ==
Datenbankabfragen werden genauso wie Änderungen konstruiert - einzig die zusätzliche Methode fetch() wird danach aufgerufen.
Wir haben eingangs beim Konstruieren den Standard Fetch-Modus auf ASSOC geändert. Dadurch erhalten wir ein Array Objekt mit dem Spaltennamen als Schlüssel und dem Inhalt als Wert erhalten. Führen wir die fetch() Funktion mehrmals durch erhalten wir alle Zeilen (falls mehrere vorhanden sind).

Quellcode

```
$pdoparams = array(
':userid' => $_REQUEST['userid']
);
$sql = "SELECT *
FROM user
WHERE userid = :userid ";
$stmt = MyDB::getInstance()->prepare($sql);
$stmt->execute($pdoparams);
$row = $stmt->fetch();
print_r($row);
$row = $stmt->fetch();
print_r($row);
```

Alles anzeigen

== Mehrere Daten abfragen ==

Häufig werden einfach alle Daten abgefragt, die man per SQL String definiert. Statt der Methode fetch() benutzt man dazu die Methode fetchAll(). Die Rückgabe der Funktion ist ein normales zweidimensionales Array, über das man mit foreach iterieren kann.

Quellcode

```
$pdoparams = array(
':username' => "%".$_REQUEST['username']."%",
':minage' => $_REQUEST['minage'],
':maxage' => $_REQUEST['maxage']
);
$sql = "SELECT *
FROM user
WHERE username LIKE :username
AND age BETWEEN :minage AND :maxage";
$stmt = MyDB::getInstance()->prepare($sql);
$stmt->execute($pdoparams);
foreach($stmt->fetchAll() as $row) {
print_r($row);
}
```

Alles anzeigen

== Kommaseparierte Liste mit IN ==
MySQL und andere DBMS bieten die Möglichkeit kommaseparierte Listen mit IN zu verarbeiten.
Der SQL String um die Benutzer 1,3 und 5 zu löschen würde folgendermaßen aussehen:

Quellcode

```
DELETE FROM user
WHERE userID IN (1,3,5)
```

Um eine kommaseparierte Listen verarbeiten zu können kennt das PDO leider keine Möglichkeit.
Die Klasse MyDB wird daher um eine spezielle implode Methode erweitert:

Quellcode

```
class MyDB {
private static $escapecounter = 0;
...
static public function implode(&$pdoparams, $arr) {
$tmp = array();
foreach($arr as $val) {
$key = ':implode'.self::$escapecounter++;
$pdoparams[$key] = $val;
$tmp[] = $key;
}
return implode(',', $tmp);
}
}
```

Alles anzeigen

Das Beispiel erweitern wir wie folgt:

Quellcode

```
$pdoparams = array();
$sql = "DELETE FROM user
```

```
3. WHERE userID IN (".MyDB::implode($pdoparams, array(1,3,5)).")";
4. $stmt = MyDB::getInstance()->prepare($sql);
5. $stmt->execute($pdoparams);
```

Die Methode implode() liefert einen String mit Platzhaltern zurück:

Quellcode

```
1. DELETE FROM user
2. WHERE userID IN (:implode0,:implode1,:implode2)
```

Zusätzlich wird das als Referenz übergebene Array $pdoparams um die Inhalte erweitert. Nach Ausführung von implode sieht das Array also wie folgt aus:

Quellcode

```
1. Array (
2. [:implode0] => 1
3. [:implode1] => 3
4. [:implode2] => 5
5. )
```

Das execute() bindet diese Parameter an den String und das Statement wird sicher ausgeführt. Die Lösung ist besser als sich den String mit PHP Stringfunktionen selbst zusammen zu bauen, da das PDO nur auf diese Art vor SQL Injections schützt.

== Code Download ==
Den fertigen Code von Klasse und dem letzten Beispiel gibt es zum Download unter demo.easy-coding.de/php/pdo/download.zip.