PDO - Vorwort

1. PHP Data Objects

Das ist der vollständige Name für PDO. Und genau wie MySQL und MySQLi ist es eine Erweiterung für den Zugriff auf Datenbanken. Allerdings gibt es hier ein paar Features, die neu sind. Hinzu kommen noch ein paar gravierende Unterschiede zu den beiden anderen Erweiterungen.

2. Alte Zöpfe

... in PHP wurden bereits mit MySQLi abgeschnitten. Allerdings fehlte den Entwicklern wohl der Mut, um einen radikalen Schnitt beim Datenbankzugriff vorzunehmen. Das Ergebnis war ein ziemlich fauler Kompromiss zwischen prozeduraler und objektorientierter Zugriffsmöglichkeit. Mit PDO wollte man nun diese Schwachpunkte beseitigen, was im Großen und Ganzen auch gelungen ist.

Ganz wichtig!

PDO ähnelt in vielen Dingen MySQLi, vor allem, was die Funktionalitäten angeht. Darum sollte ihr euch vorher erst mal mit der entsprechenden Theorie zu MySQLi auseinandersetzen, bevor ihr hier weitermacht. Denn hier geht es nur um die Unterschiede. Außerdem wiederhole ich mich nicht gerne. Nur wenn es um diesen Satz geht.

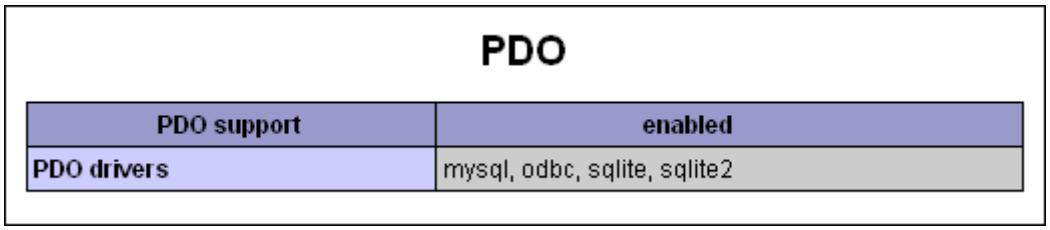
3. Die Vorraussetzungen

Sind fast identisch mit denen von MySQLi.

- PHP 5 (ab 5.1 aufwärts standardmäßig vorhanden, bei 5.0 als PECL-Erweiterung verfügbar)
- MySQL-Server 4.1 oder höher (am Besten 5.x)
- Aktivierte PDO-Unterstützung

Verfügbarkeit

Nehmt das allseits bekannte `phpinfo()`. Wenn ihr folgenden Abschnitt seht, so ist PDO verfügbar, ansonsten müsst ihr es installieren.



PDO

PDO support	enabled
PDO drivers	mysql, odbc, sqlite, sqlite2

4. Unterschiede zur MySQL- und MySQLi-Erweiterung

- PDO kann man **NUR** objektorientiert nutzen. Die rein prozedurale Vorgehensweise ist hier nicht mehr möglich.
- Verbesserte Nutzung von Prepared Statements.
- Zugriff auf unterschiedliche Datenbanksysteme.

5. Die Sache mit den unterschiedlichen Datenbanksystemen

Dieses neue Feature ist ein echter Segen. Früher benötigte man für jedes gottverdammte Datenbanksystem eine eigenen Erweiterung, um damit zu kommunizieren. Bei PDO kann man darauf endlich verzichten.

Wenn

... man die entsprechenden Treiber zur Verfügung hat. Und nun kommen wir zur ersten schlechten Meldung. Laut der aktuellen(?) Doku werden derzeit nur folgende Systeme unterstützt. Beim Rest steht das derzeitige Entwicklungsstadium immer noch auf "experimentell".

- CUBRID (was immer das sein mag)
- IBM DB2
- Informix
- MySQL
- ODBC
- PostgreSQL
- SQLite
- Microsoft SQL Server (ab Version 2005)

Wenn ihr wissen wollt, welche Datenbanksysteme unterstützt werden, so schaut euch die Ausgabe des `phpinfo()` an. In der Spalte `enabled` könnt ihr es dann sehen.

Obacht!

PDO stellt nur eine Abstraktionsschicht für den Zugriff auf unterschiedliche Datenbanksysteme dar. Sobald es um SQL-Statements geht, ist hängen im Schacht. Vulgo, PDO schafft den Zugang zu einem Datenbankserver, kümmert sich aber **NICHT** um die Abfragen.

Ein Beispiel

Wenn man auf eine MYSQL-Datenbank beim Query ein `LIMIT 0, 10` loslässt, so wird das bei einer Datenbank des MS SQL Servers unweigerlich zu einem Fehler führen, da man dort ein `TOP` benötigt. Da bedarf es einer zusätzliche Abstraktionsschicht wie zum Beispiel Doctrine.

6. Über dieses Tutorial

Ich werde hier nicht auf jeden Mäusefurz eingehen, sonder mich darauf beschränken, euch das Prinzip und die wichtigsten Möglichkeiten zu zeigen. Einen kompletten Überblick über PDO findet ihr hier.

Bitte

... arbeitet euch zuvor durch das Thema Advanced MySQL. Denn ohne dieses Wissen werdet ihr einige Punkte nicht begreifen.

PDO - Verbindungsaufbau

1. Der Konstruktor

... ist natürlich auch bei PDO vorhanden. Allerdings gibt es bei den Zugangsdaten eine Neuerung im Vergleich zu MySQL und MySQLi, denn die Angaben zum Server sind hier anders. Der Konstruktor lässt insgesamt vier Angaben zu. Das Schema sieht dabei so aus.

```
$db = new PDO([Server], [User], [Passwort], array [Optionen]);
```

2. Angaben zum Server

Konzentrieren wir uns nur mal auf die Wichtigsten. Wichtig dabei ist, dass alle Daten in einer Variablen abgelegt werden. Die Trennung erfolgt über ein Semikolon. Zuerst muss der Datenbanktyp angegeben werden. Dann kommt normalerweise der Host und anschließend die Datenbank. Die Reihenfolge der letzten beiden Angaben ist aber nicht zwingend vorgegeben.

```
$server = 'mysql:host=localhost;dbname=datenbank';
```

Erläuterung

Die Angabe des Datenbanktyps benötigt PDO für den Verbindungsaufbau. Bei einem MS SQL Server wäre das ein `mssql` und beim Oracle-Monster ein `oci`. Da ich bisher aber immer nur mit MySQL zu tun hatte, kann ich leider keine Details zum Besten geben.

Da die Angaben zum Host und zur Datenbank keine vordefinierte Reihenfolge haben, muss man mit den "Variablen" `host` und `dbname` arbeiten. Beide Bezeichnungen sind festgelegt und dürfen nicht geändert werden!

3. Der schäbige Rest

User und Passwort

Das ist genau so wie bei der MySQL- beziehungsweise MySQLi-Erweiterung und bedarf keiner weiteren Erläuterung.

Die Optionen

... sind leider datenbankspezifisch und werden daher auch als `Driver options` bezeichnet. Das bedeutet, dass sie auf den jeweiligen Typ zugeschnitten und nicht allgemein verfügbar sind. Die Details dazu findet ihr hier unter den jeweiligen Datenbanktypen. Ich konzentriere mich nur auf zwei MySQL-Beispiele, um euch das Prinzip zu zeigen. Ach ja, dabei handelt es sich um Konstanten. Das erklärt auch den Zugriff über `PDO::Konstante`.

`MYSQL_ATTR_INIT_COMMAND`

Hier kann man bereits beim Verbindungsaufbau etwas festlegen. Zum Beispiel die Zeichenkodierung.

```
array(PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8')
```

`MYSQL_ATTR_READ_DEFAULT_FILE`

Liest die Konfigurationsdatei aus. Damit können bestimmten Angaben per SQL-Injection nicht überschrieben werden.

```
array(PDO::MYSQL_ATTR_READ_DEFAULT_FILE => '/etc/my.cnf')
```

4. Ein komplettes Beispiel

... könnte so aussehen.

```
$server   = 'mysql:dbname=datenbank;host=localhost; port=3333';
$user     = 'user';
$password = 'pw';
$options  = array
            (
              PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES utf8',
              PDO::MYSQL_ATTR_READ_DEFAULT_FILE => '/etc/my.cnf'
            );
$pdo      = new PDO($server, $user, $password, $options);
```

5. Einbindung in andere Klassen

Das ist genau wie bei MySQLi überhaupt kein Problem.

```
class Db extends PDO
{

}
```

Die weitere Vorgehensweise sollte mittlerweile selbstverständlich sein. Ansonsten schaut euch mal das hier an.

PDO - Methoden

1. Übersicht

PDO besteht aus insgesamt vier Klassen. Als da wären.

- `PDO`
- `PDOStatement`
- `PDOException`
- `PDOTreiber`

Die Treiber-Klasse ignoriere ich einfach mal, da ich (fast) immer nur mit MySQL zu tun hatte und auf die Exception-Klasse gehe ich bei der Fehlerbehandlung ein. Für die eigentlichen Abfragen sind also nur `PDO` und `PDOStatement` zuständig.

2. Die Aufgabenverteilung

`PDO`

... kümmert sich um die "normalen" Abfragen. Also Sachen wie zum Beispiel `SELECT`, `INSERT` oder `DELETE`. Diese Klasse entspricht also in etwa der `MySQLi`-Klasse.

`PDOStatement`

... dagegen ist für Prepared Statements zuständig und hat somit dieselben Aufgaben wie die `MySQLi_STMT`-Klasse.

3. Aber

Diese ganze Sache mit den unzähligen `fetch_irgendwas`-Möglichkeiten wurde in `PDOStatement` verlagert. Darum gibt es auch keine `PDO_Result`-Klasse. Allerdings ist es ohne weiteres möglich, diese Methoden auch in der `PDO`-Klasse zu nutzen.

4. Die Funktionsweise

... unterscheidet sich nicht groß von MySQLi. Denn auch hier wird mit den entsprechenden Referenzvariablen gearbeitet. Es gibt dabei ein paar Feinheiten, aber auf die gehe ich peu à peu ein.

5. Unterschiede

... zu MySQLi gibt es eine ganze Menge. Ich liste mal grob die Änderungen auf.

Methoden

- `multi_query` ist rausgeflogen (warum eigentlich?).
- `real_connect` hat ebenfalls das Zeitliche gesegnet.
- Prepared Statements können viel flexibler gehandhabt werden.

Eigenschaften

- `affected_rows` wurde entfernt und durch eine Methode ersetzt. Dazu später mehr
- `insert_id` wurde auch gehimmelt und durch eine Methode ersetzt. Dazu ebenfalls später mehr

Konstanten

Bei MySQLi hatte ich immer den Eindruck, dass die mehr oder weniger überflüssig sind. Wie ein Kropf(f) halt. Aber bei PDO gibt es im Zusammenhang mit den einzelnen Methoden sehr viel und vor allem variable Möglichkeiten zur Steuerung.

Allerdings werde ich hier nur auf einige wenige eingehen. Für einen kompletten Überblick solltet ihr die Dokumentation zu Rate ziehen.

PDO - Methoden - Die PDO-Klasse

1. Wenn man sich

... Prepared Statements, Transaktionen und die Fehlerbehandlung vorerst wegdenkt, so bleiben für den "normalen" Datenbankzugriff weniger als eine Handvoll von Methoden übrig, mit denen man vernünftig arbeiten kann. Konzentrieren wir uns zunächst auf die Einfachen.

2. exec

Führt eine Abfrage aus und gibt die Anzahl der betroffenen Datensätze zurück. Diese Methode ist der Ersatz für die sonst übliche Eigenschaft `affected_rows` und sollte für `INSERT`-, `UPDATE`-, `REPLACE`- oder `DELETE`-Anweisungen genutzt werden.

```
$pdo   = new PDO (...);
$query = 'DELETE FROM blubb WHERE ID > 10';
$num   = $pdo -> exec($query);
```

3. getAvailableDrivers

Liefert ein Array aller unterstützten Datenbanktypen.

```
$pdo   = new PDO (...);
print_r($pdo -> getAvailableDrivers());
```

4. quote

Maskiert eine Zeichenkette automatisch mit Hochkommata und verkleinert damit die Gefahr einer SQL-Injection.

```
$_POST['pw'] = 'bla\' OR \'1=1';
$query = 'SELECT
            *
          FROM
            user
          WHERE
            pw = \''.$pdo -> quote($_POST['pw']).'\';
```

5. setAttribute

Damit können Standardvorgaben geändert werden. Eine komplette Liste findet ihr hier, ich selber stelle nur mal zwei als Beispiel vor. Der eigentliche Aufbau läuft nach dem Schema `[Attribut], [Wert]` ab.

Standard-Fetch-Modus

Will man den also allgemein festlegen, so geht man zum Beispiel so vor.

```
$pdo  = new PDO ($server, $user, $password);
$pdo -> setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);
```

Das entspricht in etwa einem `mysql_fetch_assoc`. Allerdings gibt es dazu ein paar Feinheiten, auf die ich im nächsten Abschnitt eingehen werde.

Rückgabewerte von Spaltennamen

Ich persönlich halte davon nicht sehr viel, aber wenn zig Entwickler es bei der Benamung besagter Tabellenspalten nicht so genau nehmen (mal komplett groß, mal komplett klein, mal ein Kuddelmuddel), so kann das hier recht hilfreich sein.

```
$pdo  = new PDO ($server, $user, $password);
$pdo -> setAttribute(PDO::ATTR_CASE, PDO::CASE_NATURAL);
```

Wichtig

Hier werden wie schon gesagt nur die Rückgabewerte geändert. Auf die eigentlichen Namen in der Datenbank hat das keinen Einfluss. Wenn man also nicht weiß, ob Spalte X nun `bla`, `Bla` oder `BLA` heißt, so kann man mit obiger Angabe sicher sein, dass der Wert im Array als Index immer ein `BLA` hat.

PDO - Methoden - Die PDO-Klasse - query

1. Suchen

Die Methode `query` ist ausschließlich für `SELECT`-Anweisungen gedacht. Für `INSERT`, `UPDATE`, `REPLACE` oder `DELETE` sollte man die schon erwähnte `exec`-Methode nutzen.

Der Rückgabewert

Die `query`-Methode gibt ein `PDOStatement`-Objekt(!) zurück. Und das ist vor allem für Anfänger ein Problem, da sie laut Doku eigentlich zur `PDO`-Klasse gehört. Und darum existieren auch verschiedene Möglichkeiten, um die Ergebnisse zu verarbeiten.

2. Standardnutzung

```
$pdo   = new PDO ($server, $user, $password);
$query = 'SELECT bla FROM blubb WHERE ID > 600';
foreach ($pdo -> query($query) as $row) {
    print_r($row);
}
```

Das hier läuft vom Prinzip her genau so ab wie bei der MySQLi-Erweiterung.

3. Der Rückgabewert

Ein `query` liefert als Ergebnis ein Objekt der `PDOStatement`-Klasse zurück. Jetzt werden sich sicher einige fragen, was der Blödsinn soll. Nun das ist ganz einfach zu erklären. Denn damit kann man die `fetch`-Methoden dieser Klasse nutzen.

Allerdings wurde gründlich aufgeräumt und von den 8 Methoden bei MySQLi sind hier nur noch 4 übrig geblieben. Außerdem zeige ich euch hier nur das Prinzip anhand eines Beispiels. Die Details kommen erst im Abschnitt über die `PDOStatement`-Klasse.

4. Zugriff auf `fetchIrgendwas`

Dazu benötigt man ähnlich wie bei MySQLi eine Referenzvariable, um die entsprechenden Methoden nutzen zu können.

```
$pdo    = new PDO ($server, $user, $password);
$query  = 'SELECT bla, blubb FROM blubber WHERE ID > 601';
$stmt   = $pdo -> query($query);
$result = $stmt -> fetchAll();
print_r ($result);
```

Erläuterung

Das Prinzip funktioniert genau wie bei MySQLi. In diesem Fall speichert man die Referenz auf den Rückgabewert von `query`, also einem `PDOStatement`-Objekt, in einer Variablen ab. Und über die greift man dann auf die entsprechenden(!) `fetch`-Methoden zu.

Das Ergebnis

... ist ein Array, in dem jeder Datensatz noch mal als numerisches und alphanumerisches Array abgespeichert worden ist. Daraus folgern wir was? Ganz einfach, ohne Parameter beim `fetchAll` arbeitet diese Methode genau wie `fetch_all` von MySQLi und einem `mysql_fetch_all` der MySQl-Erweiterung. Gottlob kann man das wunderbar steuern, und zwar über Parameter. Aber dazu komme ich später.

PDO - Transaktionen

1. Eine gute Nachricht

Im Vergleich zu MySQLi hat es bei PDO eine entscheidende Verbesserung gegeben. Denn man spart sich diesen doch ziemlich verkrampften Umweg über `autocommit` oder `BEGIN`. Ansonsten ist alles beim Alten geblieben. Ist ja auch kein Kunststück bei so einer einfachen Sache.

2. Eine Transaktion durchführen

Das Prinzip solltet ihr ja mittlerweile kennen, starten, ausführen oder zurückrudern. Zuerst leitet man die Transaktion ein.

```
$pdo = new PDO(...);
$pdo -> beginTransaction();
```

Dann werden die entsprechen SQL-Statements ausgeführt.

```
$query = 'UPDATE bla SET blubb = 1 WHERE id < 10';
$nums  = $pdo -> exec($query);
```

Und zum Schluss wird die Transaktion entweder ausgeführt

```
if (Bedingung) {
  $pdo -> commit();
}
```

oder zurückgesetzt.

```
else {
  $pdo -> rollBack();
}
```

3. Wirkungsweise

Im Gegensatz zu MySQLi gibt es hier nur eine Art, wie das umgesetzt wird. Nämlich über `autocommit`. Dabei passiert das hier.

`beginTransaction`

Diese Methode setzt `autocommit` automatisch auf `false`. Es wird also **NICHT** mit einer SQL-Abfrage à la `BEGIN` gearbeitet.

`commit` / `rollBack`

Dise beiden Methoden ändern dann den Wert für `autocommit` wieder auf `true` und alles weitere funktioniert wieder wie gehabt.

4. Der Vorteil

Wie ich schon sagte, muss man bei MySQLi darauf achten, wie man die Transaktion startet. Bei PDO ist das ziemlich lattens, da alles automatisch abläuft. Und meiner Meinung nach ist das ein echtes Plus.

PDO - Methoden - Die PDOStatement-Klasse

1. Eine gute Nachricht

Bei dieser Klasse haben sich die PHP-Entwickler mal so richtig ins Zeug gelegt und viele Verbesserungen vorgenommen, die einem die Arbeit wirklich erleichtern.

Die schlechte Nachricht

... ist, dass man sich im Vergleich zu MySQLi sehr stark umorientieren muss, eben weil vieles anders ist als bei MySQli. Aber keine Angst, das schaffen wir schon.

2. Prepared Statements ausführen

Am grundsätzlichen Prinzip hat sich natürlich nichts geändert. Die Reihenfolge ist dieselbe.

- SQL-Statement mit Platzhaltern vorbereiten
- Platzhalter mit Werten versehen
- Statement ausführen

3. Vorbereiten

Das geschieht wie gehabt mit `prepare`. Auch an der Referenzvariablen hat sich nichts geändert.

```
$pdo  = new PDO($server, $user, $password);
$query = '...';
$stmt  = $pdo -> prepare($query);
```

Genau wie `query` liefert auch `prepare` ein Objekt der `PDOStatement`-Klasse zurück. Und über diese Variable steuern wir dann die entsprechenden Methoden an.

4. Werte setzen

Hier hat sich im Vergleich zu MySQLi so viel getan, dass ich auf die Details erst im kommenden Abschnitt eingehe. Vom Prinzip her läuft das nach wie vor so.

```
$pdo   = new PDO($server, $user, $password);
$query = 'SELECT bla FROM blubb WHERE id > [Platzhalter]';
$stmt  = $pdo -> prepare($query);
$stmt -> bindFunktion([Platzhalter], [Variable], [TYP]);
```

Für die Merkbefreiten

Die `bindFunktion` gibt es natürlich nicht. In diesem Beispiel ist das nur ein Alias für die Methoden, die tatsächlich existieren. Aber wie schon gesagt, das kommt später.

5. Ausführung

Das geschieht wie gehabt mittels der Methode `execute`. Ein komplettes Prepared Statement sieht also prinzipiell so aus.

```
$pdo   = new PDO($server, $user, $password);
$query = 'SELECT bla FROM blubb WHERE id > ?';
$stmt  = $pdo -> prepare($query);
$stmt -> bindFunktion([Platzhalter], [Variable], [TYP]);
$stmt -> execute();
```

PDO - Methoden - Die PDOStatement-Klasse - Platzhalter setzen

1. Die grundsätzlichen Möglichkeiten

Im Gegensatz zu MySQLi hat man hier die Wahl! Man kann also die Platzhalter durch die Reihenfolge setzen oder "eigene Variablen" festlegen.

2. Über die Reihenfolge

Im Query geschieht das mit dem (hoffentlich) schon bekannten Fragezeichen.

```
$query = 'INSERT INTO bla (blubb, blubber) VALUES (?, ?)';
```

Das Ersetzen erfolgt dann wie gehabt über die Reihenfolge.

```
$blubb    = 1;
$blubb4er = 'blubber';
$query    = 'INSERT INTO bla (blubb, blubber) VALUES (?, ?)';
...
$stmt    -> bindFunktion(1, $blubb, [Typ]);
$stmt    -> bindFunktion(2, $blubber, [Typ]);
```

3. Über "eigene Variablen"

In diesem Fall arbeitet man nicht mit dem Fragenzeichen sondern mit einer konkreten Angabe, die mit einem Doppelpunkt beginnt.

```
$laber   = 1;
$schwall = 'jodelbla';
$query   = 'INSERT INTO bla (blubb, blubber) VALUES (:blubb, :blubber)';
...
$stmt   -> bindFunktion(':blubb', $laber, [Typ]);
$stmt   -> bindFunktion(':blubber', $schwall, [Typ]);
```

4. Die Fehlerbehandlung

... ist in diesem Fall leider ganz mies gelöst worden. Oder genauer gesagt, sie existiert praktisch nicht. Denn es wird nur geprüft, ob der Platzhalter im Query mit dem ersten Wert der `bindFunktion` übereinstimmt. Und das war es schon.

Beispiele

```
$query   = 'INSERT INTO bla (blubb) VALUES (:blubb)';
$stmt   -> bindFunktion('#blubb', $laber);
```

Bei dieser Kombination fliegt dem geneigten Betrachter eine Meldung à la `Warning: PDOStatement::execute() [pdostatement.execute]: SQLSTATE[HY093]: Invalid parameter number: parameter was not defined ...` um die Ohren, während bei den beiden folgenden Varianten nichts, aber auch gar nichts passiert. Außer, dass man kein Ergebnis bekommt.

```
$query   = 'INSERT INTO bla (blubb) VALUES (#blubb)';
$stmt   -> bindFunktion(':blubb', $laber);

$query   = 'INSERT INTO bla (blubb) VALUES (#blubb)';
$stmt   -> bindFunktion('#blubb', $laber);
```

Wenn man so was vermeiden will, muss man prüfen, ob der Rückgabewert der `bindFunktion` ein `true` oder `false` ist.

```
$query = 'INSERT INTO bla (blubb) VALUES (#blubb)';
if($stmt -> bindFunktion(':blubb', $laber)) {
  ...
}
else {
  ...
}
```

PDO - Methoden - Die PDOStatement-Klasse - Daten aufbereiten

1. Die Methoden

Um denn nun die entsprechenden Platzhalter umzuwandeln, bietet uns PDO drei verschiedene Methoden an. Als da wären.

bindValue

Diese Methode arbeitet praktisch genau so wie `bind_param` von MySQLi.

```
$blubb    = 1;
$blubber  = 'blubber';
$query    = 'INSERT INTO bla (blubb, blubber) VALUES (?, :blubber)';
...
$stmt    -> bindValue(1, $blubb, [Typ]);
$stmt    -> bindValue(':blubber', $blubber, [Typ]);
```

bindParam

Hier gelten dieselben Regeln wie bei `bindValue`. Es gibt nur einen Unterschied. Die Parameter sind zunächst mal nur Referenzen, die erst bei einem `execute` ausgewertet werden. Was das im Detail soll, weiß ich auch noch nicht ganz.

```
$blubb    = 1;
$blubb4er = 'blubber';
$query    = 'INSERT INTO bla (blubb, blubber) VALUES (?, :blubber)';
...
$stmt    -> bindParam(1, $blubb, [Typ]);
$stmt    -> bindParam(':blubber', $blubber, [Typ]);
```

bindColumn

Das funktioniert in etwa wie `bind_result` von MySQLi. Denn hier kann man die Spaltennamen der Tabellen an Variablen binden. Auch hier kann man sowohl über die Reihenfolge als auch mit konkreten Namen arbeiten.

```
$query = 'SELECT bla, blubb FROM blubber';
...
$stmt  -> bindColumn(1, $laber);
$stmt  -> bindColumn('blubb', $schwall);
```

2. Typangaben

Im Gegensatz zu MySQLi hat man die Varianten natürlich mal wieder komplett umgebaut.
`PARAM_BOOL` (Boolsche Variable)

- `PARAM_INT` (Ganzzahl)
- `PARAM_NULL` (NULL-Wert von MySQL)
- `PARAM_STR` (Zeichenkette)

Bei diesen Werten handelt es sich um Konstanten, die als dritter Parameter an die jeweiligen Methoden übergeben werden können. Bei `PARAM_STR` gibt es zusätzlich die Möglichkeit, als vierten Parameter noch die Länge vorzugeben.

```
$query = 'SELECT bla, blubb FROM blubber';
...
$stmt  -> bindColumn(1, $laber, PDO::PARAM_INT);
$stmt  -> bindColumn('blubb', $schwall, PDO::PARAM_STR, 5);
```

Wichtig!

In MySQLi konnte man beim Typ `d` auch Fließkommazahlen angeben. In PDO ist das leider nicht möglich, da `PARAM_INT` nur mit Ganzzahlen klarkommt. In so einem Fall muss man leider mit `PARAM_STR` arbeiten.

PDO - Methoden - Die PDOStatement-Klasse - Daten holen

1. Die Methoden

PDO bietet vier Möglichkeiten, um sich die Daten eines Prepared Statements oder eines normalen `query` zu holen. Genau wie bei MySQLi oder der MySQL-Erweiterung wird das Ergebnis ebenfalls in einer Variable zwischengespeichert.

2. `fetchAll`

Auf diese Methode habe ich euch ja schon hingewiesen. Und sie ist mein persönlicher Liebling, da sie alle Daten in einem Rutsch holt. Das Ergebnis wird dann als Array zurückgegeben. OK, diese Zuneigung basiert auf meiner persönlichen Art zu programmieren, also sei es drum.

```
$query = 'SELECT bla, blubb FROM blubber WHERE ID > ?';
...
$stmt  -> execute();
// Holt alle Datensätze
$result = $stmt->fetchAll();
print_r($result);
```

3. `fetch`

Hiermit besorgt man sich immer nur den nächstmöglichen Datensatz als Array. Auf die Details gehe ich im folgenden Abschnitt ein.

```
$query = 'SELECT bla, blubb FROM blubber WHERE ID > ?';
...
$stmt  -> execute();
// Holt sich den nächsten Datensätze
$result = $stmt->fetch();
print_r($result);
```

4. `fetchObject`

Funktioniert genau so wie `fetch`, nur dass das Ergebnis als Objekt zurückkommt. Auf die Details gehe ich ebenfalls erst im folgenden Abschnitt ein.

```
$query = 'SELECT bla, blubb FROM blubber WHERE ID > ?';
...
$stmt  -> execute();
// Holt sich den nächsten Datensätze als Objekt
$result = $stmt->fetchObject();
print_r($result);
```

5. `fetchColumn`

Was diese Methode bezwecken soll, ist mir heute noch ein Rätsel. Damit holt man sich vom nächstmöglichen Datensatz den Wert einer Spalte. Und bei der Auswahl, kann man nur mit Zahlen arbeiten. Also `0` für die erste Spalte des SQL-Statements, `1` für die nächste und so fort.

```
$query = 'SELECT bla, blubb FROM blubber WHERE ID > ?';
...
$stmt  -> execute();
// Holt sich von nächsten Datensatz das Feld blubb
$result = $stmt->fetchColumn(1);
print_r($result);
```

Wenn ihr mich fragt, so lasst die Finger von dieser Methode. Die ist Humbug. Zum Feintuning der anderen komme ich jetzt.

PDO - Methoden - Die PDOStatement-Klasse - Daten verarbeiten

1. Parameter

Wenn wir mal diesen `fetchColumn`-Blödsinn außer Acht lassen, so kann man das Ergebnis einer `fetchIrgendwas`-Methode über ein paar nützliche Konstanten steuern. Leider gibt es da massig Kombinationsmöglichkeiten. Manche Dinge sind hier erlaubt und da nicht. Oder es ist anders herum.

Daher solltet ihr euch die Möglichkeiten zur Beeinflussung des Rückgabewertes bei den entsprechenden Methoden genau zu Gemüte führen. Und natürlich auch ein wenig herum spielen, da leider nicht alles ausführlich dokumentiert wurde. Ich selber stelle euch nur ein paar Beispiele vor.

2. Standardmöglichkeiten

FETCH_NUM

Das entspricht einem `mysql_fetch_num` der MySQL-Erweiterung. Das Ergebnis ist ein numerisches Array. Diese Konstante kann bei `fetch` und `fetchAll` verwendet werden.

```
$result = $stmt -> fetchAll(PDO::FETCH_NUM);
```

FETCH_ASSOC

Das entspricht einen `mysql_fetch_assoc`. Das Ergebnis ist ein Array mit den Spaltennamen der Tabelle als Index. Diese Konstante kann ebenfalls bei `fetch` und `fetchAll` verwendet werden.

```
$result = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

FETCH_BOTH

Das entspricht einen `mysql_fetch_array`. Dabei handelt es sich um eine Kombination von `FETCH_NUM` sowie `FETCH_ASSOC`. Die unterstützten Methoden sind identisch. Wenn man keinen Parameter angibt und auch nicht mit `setAttribute` gearbeitet hat, so ist dies die Standardvorgabe.

```
$result = $stmt->fetchAll(PDO::FETCH_BOTH);
```

FETCH_OBJ

Das entspricht einen `mysql_fetch_object`. Das Ergebnis wird als Objekt zurückgegeben,

```
$result = $stmt->fetchAll(PDO::FETCH_OBJ);
```

3. Spezielle Fälle

FETCH_COLUMN

Hiermit lässt man sich nur eine Spalte ausgeben. Der zweite Parameter bestimmt die Zahl des Feldes. Diese Konstante kann bei `fetch` und `fetchAll` verwendet werden.

```
$query  = 'SELECT bla, blubb, blubber FROM laber';
...
// Holt nur Spalte blubb
$result = $stmt->fetchAll(PDO::FETCH_COLUMN, 1);
```

FETCH_GROUP

Nimmt die erste Spalte als Index und den Rest als Inhalt.

```
$query  = 'SELECT bla, blubb, blubber FROM laber';
...
// Nimmt bla als Index
$result = $stmt->fetchAll(PDO::FETCH_GROUP);
```

4. Kombinationen

Auch das ist möglich, allerdings nur recht begrenzt. Dabei werden die einzelnen Konstanten durch ein | getrennt. Leider ist die Doku bezüglich der Möglichkeiten nicht sehr auskunftsfreudig, so dass man einfach herumspielen muss. Auch hier gibt es nur ein paar Beispiele.

FETCH_ASSOC / FETCH_GROUP

Nimmt die erste Spalte als Index und den Rest als Array mit der Standardvorgabe FETCH_BOTH. Die Reihenfolge ist egal.

```
$query  = 'SELECT bla, blubb, blubber FROM laber';
...
// Nimmt bla als Index, Rest als alphanumerisches Array
$result = $stmt->fetchAll(PDO::FETCH_ASSOC|PDO::FETCH_GROUP);
```

FETCH_COLUMN / FETCH_GROUP

Nimmt den Spaltennamen, der in FETCH_COLUMN angegeben wurde als Index und dann nur den nächsten darauf folgenden Spaltenwert

```
$query  = 'SELECT bla, blubb, blubber FROM laber';
...
// Nimmt blubber als Index und bla ist der Inhalt
$result = $stmt->fetchAll(PDO::FETCH_GROUP|PDO::FETCH_COLUMN, 2);
```

Zusammenfassung

Mit der Kombination mehrerer Konstanten könnte man echt tolle Dinge anstellen, wenn es da nicht unzählige Einschränkungen gäbe. So ist man nur auf ein paar mehr oder weniger beschränkte Möglichkeiten begrenzt, die obendrein teilweise sehr sinnfrei sind. Ich hoffe, man arbeitet daran.

PDO - Fehlerbehandlung

1. Vorab

... eine gute Nachricht. Bei PDO hat man diese schrottige Unterscheidung zwischen `connect_error`/`connect_errno` und `error`/`errno` wieder in den Orkus befördert. Also dahin, wo dieser Blödsinn hingehört.

2. Das Prinzip

PDO kennt drei Stufen bei der Fehlerbehandlung. Gesteuert werden sie über die Konstante `ATTR_ERRMODE`, plus die entsprechende Angabe eines Wertes. Auch hierbei handelt es sich um Konstanten. Als das wären.

`ERRMODE_SILENT`

Nun, zunächst mal macht diese Konstante ihrem Namen alle Ehre. Denn bei dieser Einstellung gibt PDO keinen Mucks von sich und man muss sich die Fehlermeldungen über eine Methode holen (dazu gleich mehr).

`ERRMODE_WARNING`

Durch diesen Wert ist PDO schon ein wenig auskunftsfreudiger und haut uns zum Beispiel bei einem Fehler in der SQL-Syntax oder bei einem Zugriff auf eine nicht existente Tabelle eine Warnmeldung um die Ohren.

`ERRMODE_EXCEPTION`

Wie man aus dem Namen schon erschließen kann, bringt PDO endlich auch eine eigene Exception-Klasse. Bei dieser Einstellung wird dann immer eine Ausnahme geworfen, die erstens sehr detailliert ist und zweitens sehr leicht abgefangen werden kann. Die Details dazu kommen später.

3. Die Stufe einstellen

Zunächst mal müsst ihr wissen, dass `ERRMODE_SILENT` der Standardwert ist, wenn man nicht etwas anderes angibt. Um dieses vor allen in der Entwicklungsphase zu beseitigen muss man dem Objekt aus der `PDO`-Klasse das entsprechend mitteilen. Dazu nehmen wir die (hoffentlich) schon bekannte Methode `setAttribute`. Von Prinzip her sieht das dann so aus.

```
$pdo  = new PDO ($server, $user, $password);
$pdo -> setAttribute(PDO::ATTR_ERRMODE, PDO::[FEHLERSTUFE]);
```

Und was nun?

Sagen wir es mal so. `ERRMODE_SILENT` ist Biberkacke und sollte meiner Meinung nach überhaupt nicht benutzt werden. Wer von Exceptions keine Ahnung hat, sollte zumindest mit `ERRMODE_WARNING` arbeiten. Also so.

```
$pdo  = new PDO ($server, $user, $password);
$pdo -> setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
```

Ansonsten sollte man meiner Meinung nach immer(!) mit `ERRMODE_EXCEPTION` arbeiten. Keine Ahnung, warum das nicht die Standardvorgabe ist.

```
$pdo  = new PDO ($server, $user, $password);
$pdo -> setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

PDO - Fehlerbehandlungn - Die Fehlermethoden

1. Die Methoden

Wenn man sich wider besseres Wissens für `ERRMODE_SILENT` als Standardeinstellung entscheidet, so hat man zwei Möglichkeiten, SQL-Fehler abzufangen, nämlich `errorCode` für den Fehlercode und `errorInfo` für die Fehlermeldung inklusive Fehlercode. Allerdings gibt es ein paar Feinheiten, da beide Methoden sowohl in der `PDO`- als auch `PDOStatement`-Klasse vorhanden sind.

2. Ganz wichtig!

Beide Methoden beziehen sich nur auf das zuvor ausgeführte SQL-Statement. Sie sind also nicht allgemein verfügbar, sondern müssen immer und immer und immer wieder neu gesetzt werden. Sie verhalten sich also genau so wie `mysql_error`/`mysql_errno` beziehungsweise `mysqli_error`/`mysql_errno`.

3. Vorgehensweise

Bei normalen Abfragen

... greift man einfach auf die Referenz des `PDO`-Objektes zurück.

```
$pdo    = new PDO($server, $user, $password);
// Spalte bla existiert nicht
$query  = 'SELECT bla FROm blubb';
$result = $pdo -> query ($query),
if ($pdo -> errorInfo()) {
  print_r($pdo -> errorInfo());
}
```

Als Rückgabewert erhält man dann ein Array mit drei Einträgen.

```
Array
(
  [0] => 42S22
  [1] => 1054
  [2] => Unknown column 'bla' in 'field list'
)
```

Der erste Wert steht für den allgemein gültigen und der zweite für den datenbankspezifischen Fehlercode. Im dritten Element erfährt man dann endlich, um was es eigentlich geht.

Bei Prepared Statements

... dagegen muss man die Referenz des `PDOStatement`-Objektes nehmen.

```
$query = 'SELECT bla FROM suche WHERE id = ?'
...
$stmt  = $db -> prepare($query);
$stmt -> execute();
if ($stmt->errorInfo()) {
  ...
}
```

PDO - Fehlerbehandlungn - Die Exception-Klasse

1. Ein Segen

Fürwahr, mit dieser Exception-Klasse ist endlich mal eine vernünftige Ausnahmebehandlung möglich. Das fängt schon beim Verbindungsaufbau an, wo man nicht mit irgendwelchen `error`-Methoden von MySQLi arbeiten muss. Da reicht ein einfaches

```
try {
  $pdo  = new PDO ($server, $user, $password);
}
catch (PDOException $e) {
  ...
}
```

2. Fehlerhafte SQL-Statements abfangen

Hier arbeitet man genau so, wenn man zuvor den entsprechenden Debug-Wert mittels `$pdo -> setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);` gesetzt hat. Ansonsten funktioniert das nicht.

```
try {
  // Spalte bla existiert nicht
  $query  = 'SELECT bla FROM suche WHERE id > 600';
  $result = $pdo -> query($query);
}
catch (PDOException $e) {
  print_r($e);
}
```

3. Die Fehlermeldung

... selber stellt sich bei einem `print_r` so dar. Aus Gründen der Übersicht stelle ich nur mal die wichtigen Dinge vor.

```
PDOException Object
(
  [message:protected] => SQLSTATE[42S22]: Column not found:
                         1054 Unknown column 'bla' in 'field list'
  [code:protected] => 42S22
  [file:protected] => [absoluter Pfad zur Datei]
  [line:protected] => [Fehler in Zeile X]
  [errorInfo] => Array
    (
      ...
    )
)
```

Die einzelnen Punkte

- `message` Fehlermeldung mit datenbankspezifischen Fehlercode
- `code` Offizieller Fehlercode nach SQL ANSI
- `file` Datei, die den Fehler hervorruft
- `line` Zeile, die den Fehler hervorruft
- `errorInfo` Dieselben Werte wie bei der Methode `errorInfo`

Natürlich

... kann man auch hier die üblichen Methoden der "normalen" Exception-Klasse nutzen, Also zum Beispiel `getTrace` für die Ablaufverfolgung.

4. Eine eigene Exception-Klasse?

So was kann man natürlich auch machen, wenn sie auf der `PDOException` aufbaut. Je nach Art und Umfang eines Projektes ist das sogar empfehlenswert.

```
class myPDOException extends Exception {
  ...
}
```

Bei den geplanten Tutorials werde ich darauf genauer eingehen. Und mich genauer mit den Vor- und Nachteilen befassen. Bis dahin seid geduldig und spielt ein wenig mit PDO herum.