# Einführung in PDO

(Auszüge aus "PHP und MySQL und PDO" von Marc Remolt)

PDO steht für **PHP Data Objects** und stellt für alle von PHP unterstützten Datenbanken eine einheitliche Schnittstelle zur Verfügung. Das bedeutet, dass es nicht mehr für eine Aufgabe eine Funktion pro Datenbank gibt, sondern nur noch eine einzige Funktion, besser gesagt Methode, für alle Datenbanken. Das vereinfacht die Programmierung mit mehreren Datenbanken ungemein. Daher ist PDO inzwischen die von den PHP-Entwicklern offiziell empfohlene Methode, in PHP auf Datenbanken zuzugreifen.

Ein weiterer Bonus ist, dass PDO eine komplette Neuentwicklung ist und die PHP-Programmierer bei der Gelegenheit eine Menge alten Ballast abgeworfen haben. PDO verfügt über ein objektorientiertes, modernes Interface und unterstützt sogar konsequent PHP-Ausnahmen zur Fehlerbehandlung.

Ein Problem löst PDO allerdings nicht. Die verschiedenen Datenbanken verwenden zwar alle grundsätzlich SQL, haben aber alle eigene Veränderungen und Erweiterungen eingebaut. Wenn Sie für mehrere Datenbanken entwickeln, dann kann es Ihnen also trotzdem passieren, dass Sie Code mehrfach schreiben müssen.

In diesem Lernheft wird zwar PDO, aber nur in Verbindung mit der MySQL-SQL-Syntax behandelt. Auf andere Datenbanken werde ich nicht eingehen. Kenntnisse in SQL sollten Sie bereits besitzen. Es befindet sich zwar eine kleine Wiederholung im Skript, was aber kein Ersatz für eine komplette Schulung in SQL sein kann.

## Das PDO-Objekt

PDO stellt eine objektorientierte Schnittstelle zu Datenbanken zur Verfügung. Dementsprechend wird eine Datenbank auch durch ein **Objekt der Klasse PDO** repräsentiert, das durch **new** erzeugt wird.

**Beispiel:**

```php
<?php

$db = new PDO();

?>
```

**Listing:** pdo1.php

Beachten Sie, dass die Klasse PDO entgegen dem sonst üblichen Standard komplett groß geschrieben wird, da es sich um eine Abkürzung handelt.

## Der DSN

Wenn Sie diesen Code ausführen, erhalten Sie von PHP eine Warnung, dass dem Konstruktor

mindestens ein Parameter übergeben werden muss. Bei diesem handelt es sich um den sogenannten **DSN** (**Data Source Name**), der beschreibt, wo die Datenbank zu finden ist. Dieser wird in Form eines **URI4** angegeben. Die Form ist für jede der Datenbanken spezifisch.

Für MySQL besteht er aus folgenden Elementen:

**Datenbank-Typ**: Der Typ der Datenbank, mit der sich PDO verbinden soll. Für MySQL ist das immer der String ***mysql***.

**host**: Hier können Sie den Hostnamen des Servers angeben, mit dem Sie sich verbinden wollen. Wenn sich die Datenbank auf dem gleichen Rechner wie der Webserver beindet, können Sie hier *localhost* verwenden.

**port**: Der TCP/IP-Port, auf dem die Datenbank lauscht. Der Standard-Port von MySQL ist *3306*. Dieser Parameter kann weggelassen werden.

**dbname**: Der Name der Datenbank, mit der Sie sich verbinden wollen.

### Beispiel:

Um sich also mit einer Datenbank namens *mysql* 5 auf dem lokalen Rechner zu verbinden, würde folgender Code prinzipiell funktionieren:

```
?php

$db = new PDO('mysql:host=localhost;dbname=mysql;port=3306');

?>
```

**Listing:** pdo2.php

Wie gesagt, *port* dürfen Sie auch weglassen. Wenn Sie diesen Code ausführen, wird aber trotzdem von PHP ein Fehler gemeldet:

```
PHP Fatal error: Uncaught exception
'PDOException' with message 'SQLSTATE[28000] [1045] Access
denied for user
'www-data'@'localhost' (using password: NO)' in code/lektion3/
pdo2.php:2 Stack
trace: #0 code/lektion3/pdo2.php(2):
PDO->__construct('mysql:host=loca...') #1
{main} thrown in code/lektion3/pdo2.php on line 2
```

Die exakte Meldung, genauer der Benutzername, kann sich von System zu System unterscheiden, aber die Aussage bleibt die selbe. Der genannte Benutzer hat keinen Zugriff auf die Datenbank. Das liegt daran, dass die von PHP standardmäßig verwendeten Login-Daten leider von MySQL nicht akzeptiert werden.

MySQL ist eine Datenbank, die einen Login erfordert, bevor sie Zugang zu Ihren Datenbanken gewährt. Daher müssen Sie dem Konstruktor von PDO noch einen Benutzernamen und ein Passwort als weitere, optionale Parameter übergeben. In den meisten Entwicklungsumgebungen ist das der Benutzer ***root*** mit leerem Passwort.

Achten Sie darauf, dass diese Daten weitere PHP-Parameter und nicht Teil des **DSN** sind.

**Beispiel:**

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=mysql;port=3306', 'root', '');

?>
```

**Listing:** pdo3.php

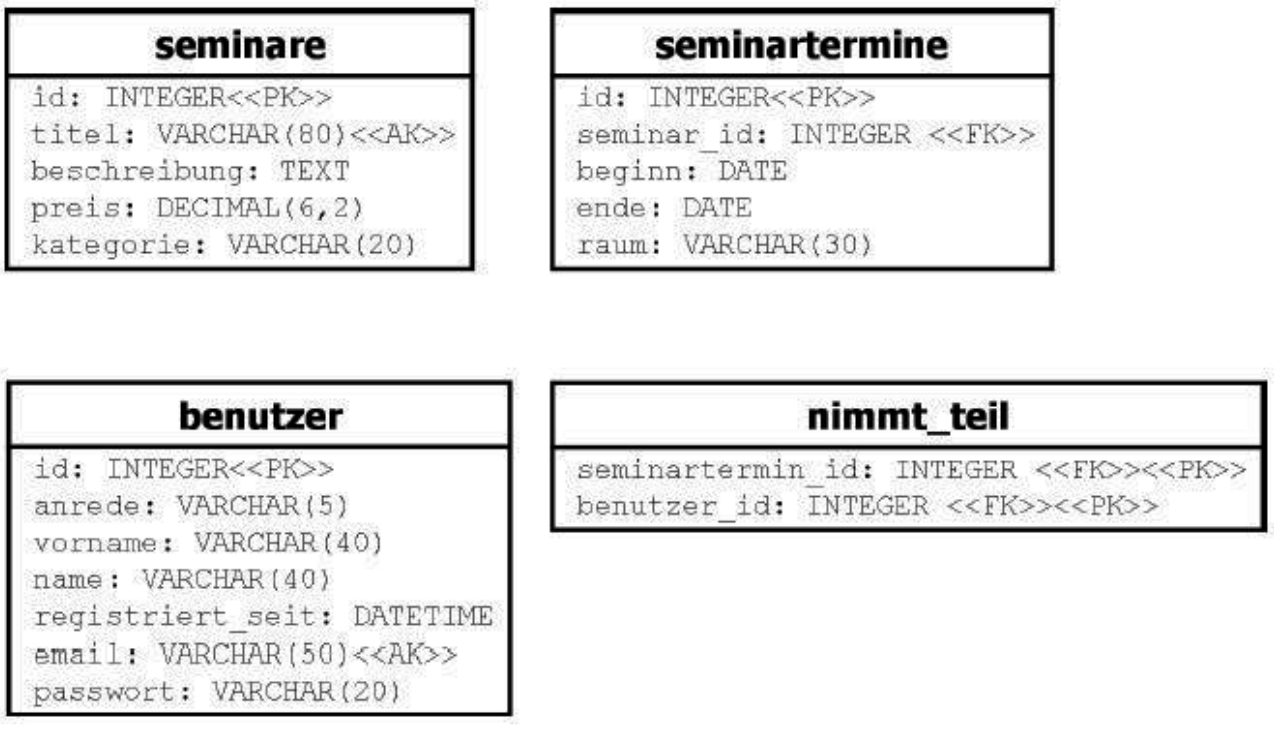
Jetzt sollte das PHP-Skript laufen, ohne einen Fehler zu erzeugen.

## Die Test-Datenbank seminarverwaltung

Falls Sie das Lernheft Datenbankentwicklung für Webanwendungen mit MySQL bereits durchgearbeitet haben, werden Sie die Datenbank *seminarverwaltung* bereits kennen.

Im Verlauf des Lernheftes haben Sie diese Datenbank, die eine Verwaltung von Seminaren, Terminen und Teilnehmern simuliert, Stück für Stück aufgebaut. Mit dieser Datenbank werden wir nun weiterarbeiten, indem wir die SQL-Abfragen in PHP einbinden. Ein praktisches Ziel dieses Lernhefts ist, am Ende ein funktionierendes Webinterface zur Administration der Seminare und Termine zur Verfügung zu haben.

Zur Wiederholung, oder zur Einführung, sehen Sie hier die Struktur von *seminarverwaltung* als physisches Datenbankmodell:



# SQL-Anweisungen mit PDO ausführen

## PDO::query()

Ein PDO-Objekt, also das, was von ***new PDO()*** erzeugt wird, verfügt über mehrere Methoden, um SQL-Anweisungen an die konfigurierte Datenbank zu senden. Im weiteren Verlauf werden Sie mehrere kennenlernen, beginnen werden wir aber mit der Methode ***PDO::query().***

Diese Methode akzeptiert als einzigen Parameter einen String mit SQL, das dann an die Datenbank gesendet wird.

**Beispiel:**

```
<?php
$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$db->query('INSERT INTO seminare (titel, beschreibung, preis, kategorie)
VALUES ("PDO", "Seminar über PHP, PDO und MySQL", 600, "programmierung")');

?>
```

**Listing:** query1.php

Da Sie noch nicht gelernt haben, Datensätze auch wieder auszulesen, müssen Sie momentan leider direkt über die MySQL-Konsole prüfen, ob der *INSERT* geklappt hat.

## PDO::errorInfo()

Eventuell hat die Anweisung aus dem letzten Abschnitt nicht geklappt, es wurde also kein neuer Datensatz in die Tabelle *seminare* eingefügt. Trotzdem wurde in PHP keine Fehlermeldung ausgegeben. Falls Sie vorhin alles richtig gemacht haben, probieren Sie doch einmal folgenden Code aus:

**Beispiel:**

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$db->query('INSERT INTO senare (titel, beschreibung, preis, kategorie)
VALUES ("PDO", "Seminar über PHP, PDO und MySQL", 600, "programmierung")');

?>
```

**Listing:** query2.php

In dieses Beispiel habe ich bewusst einen Fehler eingebaut. Die Tabelle muss natürlich *seminare* heißen, nicht ***senare***. Der Datensatz wurde nicht eingefügt (bitte nachprüfen) und trotzdem hat PHP keinen Fehler gemeldet.

Um zu sehen, ob die letzte SQL-Anweisung erfolgreich war und, wenn nicht, was das Problem war, gibt es die Methode ***PDO::errorInfo().*** Diese liefert ein Array bestehend aus zwei intern verwendeten, numerischen Fehlercodes und einer für uns hilfreicheren Fehlermeldung zurück. Falls kein Fehler aufgetreten ist, enthält das Ergebnis nur ein Element mit dem Zahlencode *0000*.

**Beispiel:**

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$db->query('INSERT INTO senare (titel, beschreibung, preis, kategorie)
VALUES ("PDO", "Seminar über PHP, PDO und MySQL", 600,"programmierung")');

var_dump($db->errorInfo());

?>
```

**Listing:** error_info1.php

Das Skript liefert als Ausgabe die Meldung:

***Table 'seminarverwaltung.senare' doesn't exist***

was genau unseren Erwartungen entspricht. Eine Tabelle *senare* gibt es nicht in unserer Beispiel-Datenbank.

```
array(3) { [0]=> string(5) "42S02" [1]=> int(1146) [2]=> string(46)
"Table 'seminarverwaltung.senare' doesn't exist" }
```

Wie Sie sehen, ist das Standardverhalten von PDO, keine Fehlermeldungen auszugeben. Das mag für den produktiven Einsatz ideal sein, denn weder die Besucher Ihrer Webseiten möchten hässliche Fehlermeldungen sehen, noch wollen Sie, dass eventuell sensible Informationen (wie z. B. Tabellennamen) öffentlich sichtbar auf Ihrer Webseite stehen. Während Sie aber die Webseite lokal auf Ihrem Rechner entwickeln, wäre es besser, sofort zu sehen, ob und was schiefgelaufen ist.

## Ergebnisse verarbeiten: Die Klasse PDOStatement

Wenn eine SQL-Abfrage ohne Fehler durchgeführt wird, erhalten Sie als Rückgabewert der Methode ***PDO::query()*** ein neues Objekt, das als Klasse ***PDOStatement*** hat. Es **repräsentiert** das Ergebnis der SQL-Anweisung und Sie können es verwenden, um die Ergebnisdaten auszulesen und weiterzuverarbeiten.

**Beispiel:**

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');
$statement = $db->query('SELECT * FROM seminare');
var_dump($statement);

?>
```

**Listing:** statement1.php

Als Ausgabe des ***var_dump()*** erhalten Sie:

*object(PDOStatement)#2 (1) { ["queryString"]=> string(22) "SELECT * FROM seminare" } .*

Wie erwartet handelt es sich bei ***$statement*** um ein Objekt der Klasse ***PDOStatement***. Auffällig ist, dass Sie in der Ausgabe keine Ergebnisdatensätze sehen können, nur das Original-SQL.

Der Grund ist, das Objekt enthält nicht das Ergebnis, es repräsentiert es nur - ein kleiner, aber wichtiger Unterschied. Das Objekt hat also nicht die Datensätze vorrätig, es weiß aber, wie es an diese herankommt. Sie können das Objekt also jederzeit nach dem eigentlichen Ergebnis fragen und erhalten die Datensätze als Ergebnis.

## PDOStatement::fetchAll()

Dies erreichen Sie über die Methode ***PDOStatement::fetchAll()***, die das Ergebnis des *SELECT* in Form eines mehrdimensionalen Arrays zurückgibt.

### Beispiel:

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root', '');

$statement = $db->query('SELECT titel, preis FROM seminare');

$daten = $statement->fetchAll();

var_dump($daten);

?>
```

**Listing:** fetch_all1.php

```
array(7) { [0]=> array(4) { ["titel"]=> string(31) "Relationale Datenbanken & MySQL" [0]=> string(31)
"Relationale Datenbanken & MySQL"
["preis"]=> string(6) "975.00" [1]=> string(6) "975.00" }
[1]=> array(4) {["titel"]=> string(13) "Ruby on Rails" [0]=> string(13) "Ruby on Rails"
["preis"]=> string(7) "2500.00" [1]=> string(7) "2500.00" }
... [6]=> array(4) {["titel"]=> string(44) "Digitale Bildbearbeitung mit Adobe Photoshop" [0]=>
string(44) "Digitale Bildbearbeitung mit Adobe Photoshop"
["preis"]=> string(7) "1500.00" [1]=> string(7) "1500.00" } }
```

Ich habe die Ausgabe um einige Datensätze gekürzt, um das Beispiel nicht zu lang werden zu lassen. Wahrscheinlich ist Ihnen aufgefallen, dass jedes Attribut doppelt im Array vorkommt, einmal mit numerischem Index, einmal mit assoziativem. Das ist leider das tandardverhalten von PDO an dieser Stelle, wir werden dies aber in Abschnitt korrigieren.

Statt dem ***var_dump()*** werden wir nun die Datensätze hübsch in Form einer richtigen HTML-Tabelle ausgeben. Da es sich bei den Datensätzen um ein reguläres PHP-Array handelt, können Sie es prima mit einer ***foreach***-Schleife durchlaufen.

**Beispiel:**

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$statement = $db->query('SELECT titel, preis FROM seminare');
$daten = $statement->fetchAll();

?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html;charset=UTF-8" />
<title>Seminare</title>
</head>
<body>
<h1>Seminare</h1>
<table>
<tr>
<th>Titel</th>
<th>Preis</th>
</tr>
<?php foreach ($daten as $seminar): ?>
<tr>
<td><?php echo $seminar['titel'] ?></td>
<td><?php echo $seminar['preis'] ?>?</td>
</tr>
<?php endforeach; ?>
</table>
</body>
</html>
```

**Listing:** fetch_all2.php

Wenn man von der Tatsache absieht, dass das Array *$daten* nun aus einer MySQL Datenbank befüllt wird, dürfte das gewohntes Terrain für Sie sein.

## PDOStatement::fetch()

Hin und wieder möchten Sie nur einen einzelnen Datensatz aus der Datenbank auslesen. Zum Beispiel soll nur eine bestimmte Nachricht oder nur ein Artikel eines Produktkatalogs gezeigt werden. Sie wissen also, dass nur ein einzelner Datensatz zurückkommen wird.

Prinzipiell können Sie auch dort mit *PDOStatement::fetchAll()* arbeiten, dies hat jedoch einen kleinen Nachteil. Auch wenn nur ein Datensatz von der Abfrage zurückkommt, wird dieser trotzdem in ein zweidimensionales Array verpackt.

**Beispiel:**

```php
<?php
$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$statement = $db->query('SELECT titel, preis FROM seminare WHERE id=1 LIMIT 1');

$daten = $statement->fetchAll();
var_dump($daten);
?>
```

**Listing:** fetch_all3.php

```
array(1) { [0]=> array(4) { ["titel"]=> string(31) "Relationale Datenbanken & MySQL" [0]=> string(31)
"Relationale Datenbanken & MySQL" ["preis"]=> string(6) "975.00" [1]=> string(6) "975.00" } }
```

Um mit dem Datensatz arbeiten zu können, müssen Sie nun erst das Array auspacken. Das ist unnötiger Code und schadet der Lesbarkeit. Viel besser ist es, stattdessen die Methode ***PDOStatement::fetch()*** zu verwenden, die immer nur einen Datensatz zurückgibt. Dieser ist dafür aber auch nicht verpackt.

**Beispiel:**

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$statement = $db->query('SELECT titel, preis FROM seminare WHERE id=1 LIMIT 1');

$daten = $statement->fetch();
var_dump($daten);

?>
```

**Listing:** fetch1.php

```
array(4) { ["titel"]=> string(31) "Relationale Datenbanken & MySQL" [0]=> string(31) "Relationale
Datenbanken & MySQL" ["preis"]=>string(6) "975.00" [1]=> string(6) "975.00" }
```

Schon viel besser lesbar, oder?

Für den Fall, dass die Ergebnisdaten mehrere Datensätze enthalten, liefert ***PDOStatement::fetch()*** den ersten Datensatz zurück. Rufen Sie die Methode erneut auf, erhalten Sie den zweiten, danach den dritten usw. Das Spiel geht so lange weiter, bis alle Datensätze durchlaufen sind. Dann liefert die Methode ***false***.

**Beispiel:**

```php
<?php
$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

// hole die ersten zwei Datensätze aus der Tabelle seminare
$statement = $db->query('SELECT titel, preis FROM seminare LIMIT 2');
var_dump($statement->fetch()); // Datensatz1
var_dump($statement->fetch()); // Datensatz2
var_dump($statement->fetch()); // false

?>
```

**Listing:** fetch2.php

*array(4) { ["titel"]=> string(31) "Relationale Datenbanken & MySQL" [0]=> string(31) "Relationale Datenbanken & MySQL" ["preis"]=>string(6) "975.00" [1]=> string(6) "975.00" } array(4) {["titel"]=> string(13) "Ruby on Rails" [0]=> string(13) "Ruby on Rails" ["preis"]=>string(7) „2500.00" [1]=> string(7) "2500.00" } bool(false)*

Das erste ***PDOStatement::fetch()*** liefert den ersten Datensatz, das zweite den zweiten Datensatz und das dritte wie erwartet den booleschen Wert ***false***.

Durch dieses Verhalten können Sie die Methode auch sehr bequem in einer ***while*-Schleife** verwenden. Diese soll so lange laufen, wie ***PDOStatement::fetch()*** ein Array mit Daten liefert.

**Beispiel:**

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$statement = $db->query('SELECT titel, preis FROM seminare');

while ( $daten = $statement->fetch() ) {
var_dump($daten);
}

?>
```

**Listing:** fetch3.php

## Ergebnisse zählen

Wenn Sie wissen wollen, wie viele Ergebnisse Ihre Abfrage enthält, können Sie das mit der PHP-Funktion ***count()*** erreichen. Da die Methode ***PDOStatement::fetchAll()*** ein Array mit Ergebniszeilen zurückliefert, können Sie den Inhalt des Arrays einfach zählen.

**Beispiel:**

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$abfrage = $db->query('SELECT * FROM seminare;');

$ergebnisse = $abfrage->fetchAll();

echo count($ergebnisse);

?>
```

**Listing:** count1.php

Diese Vorgehensweise ist in Ordnung, wenn Sie die Datensätze ohnehin für diese PHP-Seite benötigen. Wenn Sie allerdings nur die Anzahl selbst brauchen, wäre es eine enorme Verschwendung von Ressourcen, alle Datensätze auszulesen und dann nur zu zählen.

Daher ist es besser, wenn Sie den SQL-Operator ***COUNT()*** direkt verwenden, da auf diese Weise nicht alle Datensätze ausgelesen werden müssen, nur um sie zu zählen. Von MySQL wird nur das Ergebnis, also die Anzahl ausgeliefert.

**Beispiel:**

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$abfrage = $db->query('SELECT COUNT(id) as anzahl FROM seminare;');
$ergebnis = $abfrage->fetch();

echo $ergebnis['anzahl'];

?>
```

**Listing:** count2.php

## Eine Abfrage schließen

Wenn Sie mit einer Abfrage fertig sind und die Daten nicht weiter benötigen, sollten Sie das ***PDOStatement*-Objekt** mit Hilfe der Funktion ***unset()*** löschen, da ansonsten unnötig Ressourcen verschwendet werden und PHP sogar in unter Umständen Schwierigkeiten hat, mehrere Abfragen gleichzeitig offen zu halten.

**Beispiel:**

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$abfrage = $db->query('SELECT * FROM seminare;');
$ergebnisse = $abfrage->fetchAll();

// sobald die Datensätze mit fetch/fetchAll ausgelesen sind, wird das

// PDOStatement-Objekt nicht länger benötigt und kann weggeräumt werden.

unset($abfrage);
echo count($ergebnisse);

?>
```

**Listing:** unset.php

Falls Sie den Code in einer Funktion oder Methode aufrufen, die ohnehin kurz darauf endet, ist das ausdrückliche Löschen der Abfrage natürlich unnötig. Mit dem Ende der Funktion/Methode wird automatisch auch die Abfrage gelöscht, da alle lokalen Variablen beim Verlassen einer Funktion weggeräumt werden.

## SQL-Wiederholung

Im nächsten Abschnitt inden Sie eine Wiederholung der wichtigsten SQL-Anweisungen und Beispiele, wie diese in PHP verwendet werden. Ich gehe davon aus, dass Sie bereits mit der Mehrzahl der besprochenen SQL-Anweisungen vertraut sind, aber eine kleine Wiederholung kann ja nie schaden.

## SELECT

Mit Hilfe der *SELECT*-Anweisung können Sie aus einer Tabelle Datensätze auslesen.

### Beispiel:

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung','root','');

$abfrage = $db->query('SELECT * FROM seminartermine LIMIT 2;');

$ergebnisse = $abfrage->fetchAll();
var_dump($ergebnisse);

?>
```

**Listing:** select1.php

Die Abfrage können Sie dann wie üblich mit ***PDOStatement::fetch()*** oder ***PDOStatement::fetchAll()*** weiterverarbeiten.

Die Methode ***PDO::query()*** akzeptiert jede Art von SQL, inklusive JOINS und anderen komplexeren SQL-Anweisungen.

### Beispiel:

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$abfrage = $db->query('SELECT st.beginn, st.ende FROM seminartermine st
JOIN seminare s ON st.seminar_id = s.id WHERE s.titel LIKE "%Datenbank%";');

$ergebnisse = $abfrage->fetchAll();
var_dump($ergebnisse);

?>
```

**Listing:** select2.php

Auch das Umbenennen von Spalten mit *AS* ist möglich, die neuen Namen finden Sie dann als Schlüssel im Ergebnis-Array wieder.

**Beispiel:**

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$abfrage = $db->query('SELECT beginn AS start, ende FROM seminartermine LIMIT 2;');

$ergebnisse = $abfrage->fetchAll();
var_dump($ergebnisse);

?>
```

**Listing:** select3.php

## INSERT

Mit einer INSERT-Anweisung können Sie einen neuen Datensatz in eine Tabelle einfügen.

**Beispiel:**

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$db->query('INSERT INTO seminartermine (beginn, ende, raum, seminar_id)
VALUES ("2010-08-12", "2010-08-25", "Schulungsraum 1", 1)');

?>
```

**Listing:** insert1.php

Wenn der Primärschlüssel in der Tabelle ***AUTO_INCREMENT*** aktiviert hat (was er unbedingt sollte) und Sie den eben von *INSERT* erzeugten Primärschlüssel wissen wollen, können Sie sich diesen bequem von der Methode ***PDO::lastInsertId()*** liefern lassen.

**Beispiel:**

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$db->query('INSERT INTO seminartermine (beginn, ende, raum, seminar_id)
VALUES ("2010-08-12", "2010-08-25", "Schulungsraum 1", 1)');

echo "Die zuletzt erzeugte ID ist: " . $db->lastInsertId();

?>
```

**Listing:** insert2.php

Der Aufruf von *PDO::lastInsertId()* wird Ihnen den Wert des Primärschlüssels von dem Datensatz zurückgeben, den Sie in Zeile 4 erzeugt haben.

## UPDATE

Mit *UPDATE* können Sie einen oder mehrere vorhandene Datensätze ändern. Welche Datensätze bearbeitet werden, hängt von der *WHERE*-Bedingung ab, die Sie verwenden. Sollten Sie das *WHERE* weglassen, so werden **alle** Datensätze geändert.

**Beispiel:**

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$abfrage = $db->query('UPDATE SEMINARE set preis=499.99 WHERE id=5');

?>
```

**Listing:** update1.php

Wenn Sie die Information benötigen, wie viele Datensätze Ihre UPDATE-Abfrage verändert hat, können Sie die Methode ***PDOStatement::rowCount()*** verwenden.

**Beispiel:**

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$abfrage = $db->query('UPDATE seminare SET preis=499.99 WHERE id=5');

echo "Es wurde " . $abfrage->rowCount() . ' Datensatz verändert.';

?>
```

**Listing:** update2.php

## DELETE

Mit der *DELETE*-Anweisung können Sie Datensätze aus einer Tabelle löschen. Welche Datensätze gelöscht werden, hängt von der *WHERE*-Bedingung ab. Alle Datensätze, auf welche die Bedingung zutrifft, werden entfernt.

***Ein DELETE ohne WHERE-Bedingung löscht also alle Datensätze dieser Tabelle.***

**Beispiel:**

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$abfrage = $db->query('DELETE FROM seminare');

?>
```

**Listing:** delete1.php

**Beispiel:**

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$abfrage = $db->query('DELETE FROM seminare WHERE id=5');

echo "Es wurde " . $abfrage->rowCount() . ' Datensatz gelöscht.';

?>
```

**Listing:** delete2.php

Mit Bedingung werden nur die gewünschten Datensätze weggeräumt, hier der Datensatz mit *id=5*.

## CREATE TABLE

Mit der Anweisung *CREATE TABLE* können Sie eine komplett neue Tabelle anlegen.

**Beispiel:**

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$abfrage = $db->query(
'CREATE TABLE mitarbeiter (
id INTEGER PRIMARY KEY AUTO_INCREMENT,
vorname VARCHAR(50),
nachname VARCHAR(50))');

?>
```

**Listing:** create_table1.php

## ALTER TABLE

Mit *ALTER TABLE* können Sie die Struktur einer Tabelle verändern, z. B. Felder umbenennen, erlaubte Längen ändern oder ein Feld von ***NULL*** auf ***NOT NULL*** setzen.

**Beispiel:**

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$db->query('ALTER TABLE mitarbeiter ADD email VARCHAR(150)');

?>
```

**Listing:** alter_table1.php

## TRUNCATE TABLE

Mit der Anweisung *TRUNCATE TABLE* können Sie eine Tabelle komplett leeren. Alle Datensätze werden gelöscht. Der Unterschied zu *DELETE FROM* ohne *WHERE* ist, dass auch Dinge wie der Autoinkrement- Wert zurückgesetzt werden. Die Tabelle wird tatsächlich wieder in einen jungfräulichen Zustand versetzt.

**Beispiel:**

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$db->query('TRUNCATE TABLE mitarbeiter');

?>
```

**Listing:** truncate_table1.php

## DROP TABLE

Mit *DROP TABLE* schließlich können Sie eine Tabelle komplett löschen. Sollten sich in der Tabelle noch Datensätze befinden, sind diese ebenfalls verloren. Wenn Sie nicht sicher sind, ob die Tabelle schon existiert und Sie diese nur löschen wollen, falls Sie existiert, können Sie die SQL-Anweisung um ***IF EXISTS*** erweitern. Auf diese Weise erhalten Sie keinen Fehler, wenn die Tabelle nicht existiert.

**Beispiel:**

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$db->query('DROP TABLE IF EXISTS mitarbeiter');

?>
```

**Listing:** drop_table1.php

## PDO-Attribute

Die Datenbank-Verbindung, also das *PDO*-Objekt, kann mit verschiedenen Attributen an Ihre speziellen Bedürfnisse angepasst werden. Die Schreibweise der einzelnen Attribute ist zwar für

Neulinge der objektorientierten Programmierung – vorsichtig ausgedrückt - gewöhnungsbedürftig, folgt dafür aber einem festen Schema.

Der Name eines Attributs wird komplett in Großbuchstaben geschrieben und besteht aus dem **Präfix *PDO*, zwei Doppelpunkten *::*** und dem eigentlichen Namen, z. B. ***ATTR_PERSISTENT***. Der vollständige Name lautet also ***PDO::ATTR_PERSISTENT***.

Für diejenigen, die schon etwas Erfahrung mit OOP haben, es handelt sich hierbei um sogenannte **Klassenkonstanten**. Es gibt also die Konstante ***ATTR_PERSISTENT*** in der Klasse ***PDO***. Die zwei Doppelpunkte markieren die Trennung zwischen Klassennamen und Konstante.

Es gibt eine ziemlich große Auswahl dieser Attribute, mit denen Sie so ziemlich jedes Verhalten von PDO beeinflussen und sehr viele Informationen auslesen können. Im Folgenden möchte ich Ihnen zuerst die beiden Möglichkeiten vorstellen, Attribute zu setzen und Ihnen danach die Attribute zeigen, die Sie - wahrscheinlich - am häufigsten benötigen werden.

## Aufgaben zur Selbstkontrolle

### Aufgabe 1:
Schreiben Sie ein PHP-Skript *eintragen.php*, das eine Tabelle namens *filme* mit den Spalten *id*, *titel*, *beschreibung* und *dauer* anlegt. Die Spalte *id* ist der Primärschlüssel.

### Aufgabe 2:
Erweitern Sie das Skript, so dass es zu Beginn immer die Tabelle löscht.

### Aufgabe 3:
Erweitern Sie das Skript, so dass es drei Datensätze in die Tabelle *filme* einträgt. Die Filme dürfen Sie frei wählen.

### Aufgabe 4:
Erstellen Sie ein neues PHP-Skript ***filme_anzeigen.php***, das die in der Tabelle ***filme*** vorhandenen Datensätze als HTML-Tabelle auslistet.

### Aufgabe 5:
Erweitern Sie das Skript ***eintragen.php***, so dass es eine Tabelle namens ***regisseure*** mit den Spalten ***id***, ***vorname*** und ***nachname*** anlegt. Die Tabelle ***filme*** erhält eine neue Spalte ***regisseur_id***, das einen Fremdschlüssel zur Tabelle ***regisseure*** (Spalte *id*) darstellt.

### Aufgabe 6:
Erweitern Sie das Skript ***eintragen.php***, so dass es die passenden Regisseure zu den Filmen in die Datenbank schreibt. Vergessen Sie nicht, die Filme mit der passenden ***regisseur_id*** zu versehen.

### Aufgabe 7:
Erweitern Sie ***filme_anzeigen.php***, so dass der volle Name des Regisseurs (also die Inhalte der Spalten ***vorname*** und ***nachname***) in einer Spalte Regisseur mit angezeigt wird.

### Aufgabe 8:
Erweitern Sie ***filme_anzeigen.php***, so dass der Name des Regisseurs anklickbar ist und auf eine neue Seite ***regisseur_anzeigen.php*** führt. Dort soll der Regisseur mit vollem Namen angezeigt werden.

### Aufgabe 9:
Erweitern Sie ***eintragen.php*** um einige zusätzliche Filme und Regisseure, wobei einige Filme den gleichen Regisseur haben sollten.

### Aufgabe 10:
Erweitern Sie ***regisseur_anzeigen.php***, so dass die Titel aller Filme aufgelistet werden, in denen der Regisseur Regie geführt hat.

## PDO-Attribute setzen

## Beim Erzeugen des PDO-Objekts

Sie können beim Erzeugen des *PDO*-Objekts, also wenn Sie ***new PDO()*** aufrufen, als vierten Parameter ein Array mit den Attributen übergeben, die Sie ändern wollen. Zur Erinnerung: Der erste Parameter war der **DSN**, gefolgt von Benutzername und schließlich das Passwort.

**Beispiel:**

```php
<?php

$optionen = array(
PDO::ATTR_PERSISTENT => true,
PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION
);

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung','root', '', $optionen);

?>
```

**Listing:** pdo_attribute1.php

Hier werden die Attribute ***PDO::ATTR_PERSISTENT*** auf ***true*** und ***PDO::ATTR_ERRMODE*** auf den Wert ***PDO::ERRMODE_EXCEPTION*** gesetzt. Das Array ***$optionen*** wird anschließend beim Erzeugen des *PDO*-Objekts mit übergeben.

## Mit PDO::setAttribute()

Sie können Attribute auch nach dem Erzeugen einer Datenbankverbindung mit ***PDO::setAttribute()*** ändern. Dabei übergeben Sie als ersten Parameter den Namen des Attributs und als zweiten Parameter den gewünschten Wert.

**Beispiel:**

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

$db->setAttribute(PDO::ATTR_DEFAULT_FETCH_MODE, PDO::FETCH_ASSOC);

?>
```

**Listing:** pdo_attribute2.php

Hier wird nach dem Aufbau der Verbindung ***ATTR_DEFAULT_FETCH_MODE*** mittels ***PDO::setAttribute()*** auf den Wert ***PDO::FETCH_ASSOC*** gesetzt.

# Auswahl von PDO-Attributen

## Persistente Datenbank-Verbindungen (PDO::ATTR_PERSISTENT)

Das Attribut ***PDO::ATTR_PERSISTENT*** legt fest, wie PDO mit Datenbankverbindungen umgeht. Normalerweise wird die Datenbankverbindung geschlossen, wenn ein PHP-Skript endet. Sind jedoch persistente Verbindungen aktiviert, wird die Verbindung zur Datenbank offen gehalten. Das nächste PHP- Skript, das eine Datenbankverbindung anfordert, muss nun keine neue aufbauen, sondern kann die bereits vorhandene nutzen.

Der Vorteil von persistenten Verbindungen besteht darin, dass der Aufbau einer Verbindung zu Datenbanken ein relativ aufwändiger Vorgang ist und Sie somit einiges an Rechenzeit sparen können, wenn Sie Verbindungen wiederverwerten. Heutzutage sind persistente Datenbankverbindungen für die allermeisten Hosting-Anbieter kein Problem und Sie können sie getrost einschalten.

Persistente Verbindungen werden aktiviert, indem Sie das Attribut ***PDO::ATTR_PERSISTENT*** auf ***true*** setzen.

**Dieses Attribut können Sie nicht mittels *PDO::setAttribute()* verändern.**

Es kann nur beim Erzeugen des Objekts eingestellt werden.

**Beispiel:**

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root', '',
array(PDO::ATTR_PERSISTENT => true));

?>
```

**Listing:** pdo_attribute3.php

## Fehlerverhalten einstellen (PDO::ATTR_ERRMODE)

Wann immer Sie eine fehlerhafte SQL-Anweisung abschicken, wird MySQL einen Fehler melden. Dieser wird jedoch standardmäßig von PDO nicht angezeigt. Das ist an sich vollkommen in Ordnung, denn in einer produktiven Webseite möchten Sie nicht, dass derartige Meldungen ungefragt erscheinen und entweder Kunden verschrecken oder potentiellen Angreifern wichtige Informationen liefern.

Während Sie jedoch die Webseite entwickeln, benötigen Sie möglichst viele Informationen über alle Probleme in Ihrer Programmierung. Daher macht es Sinn, die MySQL-Fehlermeldungen anzuzeigen. Dieses Verhalten können Sie über das Attribut **PDO::ATTR_ERRMODE** einstellen.

**Es sind drei Einstellungen möglich:**

1. ***PDO::ERRMODE_SILENT***: Dies ist der Standardfall. PDO wird keine Fehlermeldungen von MySQL weiterreichen. Wenn Sie die Meldungen sehen wollen, müssen Sie diese von Hand über die Methode ***PDO::errorInfo()*** ausgeben.

2. ***PDO::ERRMODE_WARNING***: Ist diese Einstellung aktiv, wird von PDO jedes Mal eine PHP-Warnung (warning) erzeugt, wenn MySQL einen Fehler meldet. Das PHP-Skript selbst läuft weiter, nur die Meldung wird im Browser angezeigt.

3. ***PDO::ERRMODE_EXCEPTION***: Ist ***PDO::ERRMODE_EXCEPTION*** gesetzt, wird das Skript sofort mit einem PHP-Fehler (error) abgebrochen und die Meldung ausgegeben.

### Beispiel:

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

// Fehler wird von PHP nicht angezeigt
$db->query('SELEGT * FROM senare WO id=12');

$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

// aber jetzt
$db->query('SELEGT * FROM senare WO id=12');

?>
```

**Listing:** pdo_attribute4.php

## Standard-Modus für fetch (PDO::ATTR_DEFAULT_FETCH_MODE)

Wenn Sie normalerweise mit den Methoden ***PDOStatement::fetch()*** oder ***PDOStatement::fetchAll()*** Datensätze auslesen, werden diese als kombiniertes assoziatives und numerisches Array zurückgeliefert. Für den - sehr häufig vorkommenden - Fall, dass Sie nicht beide Versionen der Daten benötigen, können Sie sich auch entweder nur das assoziative oder das numerische Array zurückgeben lassen.

Dies können Sie über das Attribut ***PDO::ATTR_DEFAULT_FETCH_MODE*** einstellen. Es existiert eine recht große Anzahl an möglichen Werten, die häufigsten drei möchte ich Ihnen vorstellen:

1. ***PDO::FETCH_BOTH***: Dies ist die Standard-Einstellung. Sie bewirkt, dass von nun an jede Datenbank-Spalte im Ergebnis-Array sowohl mit ihrem numerischen Index als auch mit ihrem Namen auftaucht.

2. ***PDO::FETCH_NUM***: Mit dieser Einstellung werden nur noch die numerischen Indizes zurückgeliefert.

3. ***PDO::FETCH_ASSOC***: Mit dieser Einstellung wird nur noch das assoziative Array mit den Spaltennamen als Indizes zurückgeliefert. Diese Einstellung möchte ich Ihnen als sinnvollen Standard wärmstens ans Herz legen.

**Beispiel:**

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root', '',
array(PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC));

$abfrage = $db->query('SELECT titel, preis FROM seminare LIMIT 2');

var_dump($abfrage->fetchAll());

?>
```

**Listing:** pdo_attribute5.php

Diese Einstellung sorgt dafür, dass standardmäßig Ergebnis-Arrays nur noch mit assoziativen Indizes befüllt werden.

Wenn Sie für einzelne Abfragen vom Standard-Fetch-Modus abweichen wollen, können Sie den gewünschten Modus einfach den Methoden ***PDOStatement::fetch()*** oder ***PDOStatement::fetchAll()*** als ersten Parameter übergeben.

**Beispiel:**

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root', '',
array(PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC));

$abfrage = $db->query('SELECT titel, preis FROM seminare LIMIT 2');

var_dump($abfrage->fetchAll(PDO::FETCH_NUM));

?>
```

**Listing:** pdo_attribute6.php

Der Standardfall bleibt weiterhin bei der alten Einstellung ***PDO::FETCH_ASSOC***, nur für diesen einen Aufruf von ***PDOStatement::fetchAll()*** wird ***PDO::FETCH_NUM*** als Modus eingestellt.

## Gepufferte Abfragen (PDO::MYSQL_ATTR_USE_BUFFERED_QUERY)

Pro Datenbank-Verbindung kann standardmäßig immer nur eine Abfrage gleichzeitig aktiv sein. Folgender Code wird also normalerweise eine Warnung produzieren:

### Beispiel:

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

// PDO::ERRMODE_WARNING aktivieren, um die MySQL-Meldungen auch zu sehen
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
$db->setAttribute(PDO::MYSQL_ATTR_USE_BUFFERED_QUERY, false);

$abfrage1 = $db->query('SELECT * FROM seminare');
$abfrage2 = $db->query('SELECT * FROM seminartermine');

?>
```

**Listing:** pdo_attribute7.php

```
Warning: PDO::query(): SQLSTATE[HY000]: General error: 2014
Cannot execute queries while other unbuffered queries are
active. Consider using
PDOStatement::fetchAll(). Alternatively, if your code is only
ever going to run
against mysql, you may enable query buffering by setting the
PDO::MYSQL_ATTR_USE_BUFFERED_QUERY attribute. in
php_pdo_mysql/code/lektion4/pdo_attribute7.php on line 9
```

Bei aktuellen PHP-Versionen - ich konnte leider nicht herausfinden, ab welcher Version genau sich das Verhalten geändert hat - ist die Einstellung standardmäßig aktiv. Wenn Sie sich in Ihrem Code auf dieses Verhalten verlassen möchten, empfehle ich trotzdem, es explizit auf ***true*** zu setzen.

Man weiß ja nie, auf welcher PHP Version genau der Code laufen wird oder wann mal wieder eine Einstellung von den PHP-Entwicklern stillschweigend umgestellt wird.

Hier steht nicht nur eine kurze Erklärung des Problems, nämlich dass nur eine ungepufferte Anfrage gleichzeitig aktiv sein kann, sondern es werden auch zwei Lösungen präsentiert:

Zum Einen können Sie ***PDOStatement::fetchAll()*** verwenden, um alle Daten aus der ersten Abfrage zu holen und diese schließen, bevor Sie die zweite Abfrage starten.

### Beispiel:

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','');

// PDO::ERRMODE_WARNING aktivieren, um die MySQL-Meldungen auch zu sehen
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
$db->setAttribute(PDO::MYSQL_ATTR_USE_BUFFERED_QUERY, false);
```

```php
$abfrage1 = $db->query('SELECT * FROM seminare');

$daten1 = $abfrage1->fetchAll();
unset($abfrage1);

// jetzt wird kein Fehler mehr erzeugt, da das erste Statement
// bereits abgearbeitet ist
$abfrage2 = $db->query('SELECT * FROM seminartermine');

?>
```

**Listing:** pdo_attribute8.php

In Zeile 5 werden alle Datensätze aus der Abfrage geholt und in $ergebnis1 gespeichert. Die Abfrage wird schließlich in Zeile 6 geschlossen. Damit ist der Weg für die zweite Abfrage in Zeile 7 frei.

Die zweite Variante ist, einen speziellen Modus von MySQL zu verwenden, die **"buffered Queries"**, also gepufferte Abfragen.

**Dieser, nur für MySQL erhältliche Modus** sorgt dafür, dass Sie so viele Abfragen gleichzeitig stellen können, wie Sie wollen.

Um diesen Modus zu aktivieren, müssen Sie wie erwähnt das Attribut ***PDO::MYSQL_ATTR_USE_BUFFERED_QUERY*** auf ***true*** setzen oder gar nichts tun, wenn diese Einstellung als Standard gesetzt ist. Welche der beiden Varianten Sie verwenden wollen, bleibt Ihnen überlassen. Sie müssen jedoch folgendes beachten:

Die Einstellung ***PDO::MYSQL_ATTR_USE_BUFFERED_QUERY*** existiert nur für MySQL. Wenn Sie mit anderen Datenbanken (Oracle, PostgreSQL, SQLite ...) arbeiten wollen, müssen Sie mit der ersten Lösung (***PDOStatement::fetchAll() + unset()***) arbeiten, um das Problem zu umgehen.

## Vorschlag für eine Standard-PDO-Verbindung

Zum Abschluss all der Einstellungen, hier noch einmal eine Standardkoniguration in der Übersicht. Von nun an werde ich in den weiteren Code-Beispielen die Datenbankverbindung nicht mehr explizit angeben, sondern diese Datei per *require_once* einbinden.

### Beispiel:

```php
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','', array(
PDO::ATTR_PERSISTENT => true,
PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
PDO::MYSQL_ATTR_USE_BUFFERED_QUERY => true
));

?>
```

**Listing:** datenbank.php

## MySQL und Unicode

Seit Version 4.1 kann MySQL mit verschiedenen Zeichensätzen umgehen. Da sich in den letzten Jahren **Unicode** in der Programmierung immer mehr durchgesetzt hat, um eines (fernen) Tages das Durcheinander mit den Zeichensätzen zu lösen, sollten auch Sie in Ihren Webseiten von Anfang an konsequent mit Unicode arbeiten.

## Unicode-konforme Tabellen erzeugen

Um in einer MySQL-Tabelle Unicode-Daten speichern zu können, müssen Sie beim Erzeugen der Tabelle den gewünschten Zeichensatz angeben. Hinter die schließende Klammer von ***CREATE TABLE*** können Sie die Einstellung ***DEFAULT CHARSET*** schreiben, um die Tabelle auf den gewünschten Zeichensatz umzustellen.

**Beispiel:**

```
<?php

require_once 'datenbank.php';

// falls die Tabelle mitarbeiter schon existiert, wegräumen
$abfrage = $db->query('DROP TABLE IF EXISTS mitarbeiter');

$abfrage = $db->query(
'CREATE TABLE mitarbeiter (
id INTEGER PRIMARY KEY AUTO_INCREMENT,
vorname VARCHAR(50),
nachname VARCHAR(50))
DEFAULT CHARSET = utf8
');

?>
```

**Listing:** unicode1.php

Beachten Sie, dass MySQL den String *utf8* ohne Bindestrich erwartet, die Schreibweise *utf-8* würde einen Fehler erzeugen. Von diesem Zeitpunkt an ist die Tabelle personen in der Lage, Unicode-Daten aufzunehmen.

## Unicode-Daten aus MySQL auslesen

Die bloße Tatsache, dass UTF-8 für eine Tabelle aktiviert ist, ist leider noch nicht ausreichend. Wenn Sie MySQL nicht explizit mitteilen, dass eine Datenbankverbindung UTF-8 verwenden soll, werden die Daten trotzdem als Standard-Latin-1 verarbeitet.

Um MySQL wirklich auf Unicode umzustellen, können Sie entweder die Server-Koniguration umschreiben, oder nach dem Verbindungsaufbau explizit Unicode verlangen.

Lösung 1 scheitert meistens daran, dass Ihr Webhoster Sie garantiert nicht an die MySQL-Konigurationsdateien heran lässt, also bleibt Lösung 2.

Über die SQL-Anweisung ***SET NAMES*** können Sie MySQL mitteilen, dass alle Eingaben von nun an in dem gewünschten Zeichensatz kommen und auch der Server alle Daten in diesem Zeichensatz ausliefern soll.

**Beispiel:**

```
<?php

$db = new PDO('mysql:host=localhost;dbname=seminarverwaltung', 'root','', array(
PDO::ATTR_PERSISTENT => true,
PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION,
PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_ASSOC,
PDO::MYSQL_ATTR_USE_BUFFERED_QUERY => true
));

$db->query('SET NAMES utf8');

?>
```

**Listing:** datenbank_mit_unicode.php

Ab dem Aufruf von ***PDO::query()*** in Zeile 3 werden alle Daten von MySQL als UTF-8 interpretiert. In Verbindung mit den Unicode-Tabellen haben Sie nun Ihre PHP-Anwendung komplett auf Unicode umgestellt.

Ich würde vorschlagen, Sie nehmen diese ***SET NAMES*-Anweisung** in Ihre Standard-Datenbankverbindung (*datenbank.php*) auf. Auf diese Weise ist alles fertig konfiguriert und Sie müssen die Datei nur überall einbinden.

## Prepared Statements

Jeder PHP-Programmierer kennt das Problem: Es ist, vorsichtig ausgedrückt, eine Qual, SQL-Anweisungen mit PHP-Variablen zu modiizieren. Auch Sie haben wahrscheinlich schon Code wie diesen geschrieben:

**Beispiel:**

```
<?php
require_once 'datenbank_mit_unicode.php';

$titel = 'PDO-Seminar';
$beschreibung = 'Ein Seminar über PDO.';
$preis = 1249.99;
$kategorie = 'programmierung';

$db->query('INSERT INTO seminare (titel, beschreibung, preis, kategorie)
VALUES("' . $titel . '", "' . $beschreibung . '",' .$preis . ',"' . $kategorie . '")');

?>
```

**Listing:** sql_variablen_substitution.php

Die Variablen ***$titel*, *$beschreibung*, *$preis* und *$kategorie*** müssen über String-Verknüpfungen in das SQL eingefügt werden. Gerade bei Variablen, die Strings enthalten, haben wir besonders viel Spaß mit den unterschiedlichen Anführungszeichen.

Dabei ist dieses Statement, wenn wir ehrlich sind, noch relativ simpel. Größere Tabellen oder gar *SQL-JOINS* über Tabellen hinweg sorgen garantiert dafür, dass wir hier die Übersicht verlieren. Mir passiert das jedenfalls regelmäßig.

## Einführung in prepared statements

Daher möchte ich Ihnen eine der besten Erindungen der Datenbank-Programmierung vorstellen: **prepared statements**. Sie unterscheiden sich von normalen SQLAbfragen, indem das Deinieren des SQLs und die Ausführung in zwei Schritte aufgetrennt werden.

Zuerst wird mit der Methode ***PDO::prepare()*** das SQL festgelegt. Diese Methode liefert ein Ihnen bereits bekanntes ***PDOStatement*-Objekt** zurück. Danach wird das SQL mit der Methode ***PDOStatement::execute()*** ausgeführt.

### Beispiel:

```
<?php

require_once 'datenbank_mit_unicode.php';

// $statement ist mir zu lang, daher $stmt
$stmt = $db->prepare('INSERT INTO seminare (titel, beschreibung, preis,kategorie)
VALUES ("PDO-Seminar", "Ein Seminar über PDO.", 1249.99,"programmierung")');

$stmt->execute();

?>
```

**Listing:** prepared_statement1.php

Jetzt fragen Sie sich vielleicht, worin der Vorteil liegt, eine Anweisung durch zwei zu ersetzen. Und Sie haben recht, momentan bringt Ihnen die neue Schreibweise nichts.

Der Vorteil von **prepared statements** liegt darin, dass in ihnen **Platzhalter (*?*)** verwendet werden können, die Sie erst im ***PDOStatement::execute()*** mit Werten befüllen können. Diese werden der Methode als Parameter in Form eines Arrays übergeben.

### Beispiel:

```
<?php

require_once 'datenbank_mit_unicode.php';

$daten = array(
'PDO-Seminar',
'Ein Seminar über PDO.',
1249.99,
```

```
'programmierung'.
);

// $statement ist mir zu lang, daher $stmt
$stmt = $db->prepare('INSERT INTO seminare (titel, beschreibung, preis,kategorie)
VALUES (?, ?, ?, ?)');

$stmt->execute($daten);

?>
```

**Listing:** prepared_statement2.php

Die Werte in der zweiten Klammer wurden einfach durch Fragezeichen ersetzt. Beachten Sie auch, dass um die Fragezeichen keine Anführungszeichen stehen müssen.

Repräsentiert ein **?** eine Datenbank- Spalte mit Text (z.B. varchar, text ...), wird dieser automatisch in Anführungszeichen gesetzt.

Da nun aber die Werte für das *INSERT* fehlen, müssen sie beim Aufruf von ***PDOStatement::execute()*** als Parameter in Form eines numerischen Arrays nachgereicht werden. Die Werte müssen in der Reihenfolge im Array stehen, wie sie in der SQL- Anfrage stehen würden.

Sehen Sie, wie viel übersichtlicher der Code jetzt geworden ist? Keine Stringverknüpfungen mehr, keine Vermischung von Anführungszeichen - als ich zum ersten Mal mit **prepared statements** gearbeitet habe, war ich spontan begeistert!

## Mehrfaches Ausführen von prepared statements

Ein weiterer Vorteil von **prepared statements** ist, dass ein einzelnes Statement mehrmals mit ***PDOStatement::execute()*** ausgeführt werden kann. Sie müssen das eigentliche SQL nur einmal schreiben und mit ***PDO::prepare()*** vorbereiten und können es mit verschiedenen Werten so oft Sie wollen wiederverwenden.

### Beispiel:

```
<?php

require_once 'datenbank_mit_unicode.php';

// $statement ist mir zu lang, daher $stmt
$stmt = $db->prepare('INSERT INTO seminare (titel, beschreibung, preis,kategorie)
VALUES (?, ?, ?, ?)');

$stmt->execute(array('PDO-Seminar', 'Ein PDO-Seminar', 1249.99,'programmierung'));
$stmt->execute(array('PHP-Seminar', 'Ein PHP-Seminar', 1449.99,'programmierung'));
$stmt->execute(array('HTML-Seminar', 'Ein HTML-Seminar', 449.99,'design'));

?>
```

**Listing:** prepared_statement3.php

Wir haben auf dasselbe **prepared statement** drei Mal die Methode *execute()* mit verschiedenen Werten aufgerufen. Das ist im Prinzip dasselbe, wie drei Mal ein *PDO::query()* mit den kompletten Anweisungen abzusetzen - nur kürzer, und wir gewinnen als Bonus sogar noch ein wenig Geschwindigkeit. Die erhöhte Lesbarkeit des Codes sollte jedoch für Sie der wichtigste Grund sein.

Verwenden Sie **prepared statements** immer, wenn Sie Variablen in die Anweisung einbauen oder eine Anweisung mehrfach ausführen müssen. Für alle anderen Fälle ist ein einfaches ***PDO::query()*** genauso geeignet.

## Limitierungen von Platzhaltern in prepared statements

Sie können Platzhalter nur auf der Seite der Werte verwenden, niemals als Ersatz für einen Tabellen- oder Spaltennamen. Da der Ausdruck schon während der Methode ***PDO::prepare()*** auf seine Korrektheit überprüft wird (Ist die Syntax in Ordnung, existieren alle verwendeten Tabellen, existieren alle benannten Spalten ...), müssen diese Informationen schon vorliegen und können nicht erst später eingefügt werden.

Folgende Beispiele sind also in *PDO::prepare()* **NICHT** erlaubt:

***SELECT * FROM ? WHERE id = 1***
***SELECT ? FROM seminare WHERE id = 1***
***SELECT * FROM seminare WHERE ? = 1***

## Prepared statements mit benannten Platzhaltern

Statt den bereits bekannten Fragezeichen können Sie auch Namen als Parameter in **prepared statements** verwenden.

**Dies hat mehrere Vorteile:**
Benannte Platzhalter machen die SQL-Anweisung noch lesbarer. Anstatt eines Fragezeichens steht tatsächlich da, was an dieser Stelle ersetzt werden soll.

Mit benannten Platzhaltern müssen Sie *PDOStatement::execute()* ein assoziatives Array übergeben. Gerade wenn Daten schon in Form eines assoziativen Arrays vorliegen, ist es so nicht mehr notwendig, dieses Array auf ein numerisches umzubauen. Ein beliebtes Beispiel ist *$_POST* mit Formulardaten.

Bei assoziativen Arrays ist natürlich auch die Reihenfolge der Einträge nicht mehr wichtig, nur noch die Namen der Schlüssel. Sie müssen also nicht mehr darauf achten, die Parameter im Array in der korrekten Reihenfolge zu halten. Das macht Ihren Code robuster gegen Fehler.

**Anwendung:**

Benannte Platzhalter verwenden die Syntax *:platzhalter*, also z.B. *:vorname* oder *:email*. Der Doppelpunkt vor dem Namen ist für PDO das Zeichen, dass es kein gewöhnlicher String ist, sondern eben ein benannter Platzhalter. Sie dürfen ihn nicht weglassen.

Der Methode *PDOStatement::execute()* übergeben wir nun ein assoziatives Array, wobei die Schlüssel den Namen der Platzhalter entsprechen, nur ohne den Doppelpunkt. Aus *:email* wird also *$array['email']* .

**Beispiel:**

```php
<?php

require_once 'datenbank_mit_unicode.php';

$daten = array(
'titel' => 'PDO-Seminar',
'beschreibung' => 'Ein Seminar über PDO.',
'preis' => 1249.99,
'kategorie' => 'programmierung'
);

$stmt = $db->prepare('INSERT INTO seminare (titel, beschreibung, preis,kategorie)
VALUES (:titel, :beschreibung, :preis, :kategorie)');

$stmt->execute($daten);

?>
```

**Listing:** prepared_statement4.php

Falls der selbe Platzhalter im SQL mehrfach vorkommen sollte, z.B.

*SELECT * FROM seminartermine WHERE beginn > :datum OR ende > :datum*

müssen Sie den Wert (hier *:datum*) der Methode ***PDOStatement::execute()*** nur einmal übergeben.

Ansonsten gilt: Sie müssen ***PDOStatement::execute()*** exakt so viele Werte übergeben, wie Sie Platzhalter im SQL verwendet haben. Ansonsten wird PHP Ihnen das mit einer hässlichen Fehlermeldung quittieren. Wann immer Sie eine Fehlermeldung lesen, die den Text:

***number of bound variables does not match number of tokens***

enthält, stimmt die Anzahl der Platzhalter nicht mit der Anzahl der Elemente des Daten-Arrays überein.

Es ist Ihnen überlassen, mit welcher Art Platzhaltern Sie lieber arbeiten. Beide arbeiten zuverlässig und es existiert keine offiziell bevorzugte Variante. Ich unterscheide immer daran, ob die Daten als numerisches Array oder als einzelne Werte vorliegen (?-Platzhalter) oder aber ohnehin schon als assoziatives Array (benannte Platzhalter).

# PDO und Sicherheit

## SQL-Injection

Unter den Begriff **SQL-Injection** versteht man das unerlaubte Einschleusen von SQL-Anweisungen in Ihren Code durch den Besucher Ihrer Webseite. Im schlimmsten Fall kann so ein Benutzer beliebiges SQL (z.B. *DELETE FROM*, *UPDATE* ...) auf Ihrem Server ausführen. Erschreckenderweise geht das leichter als Sie denken. Betrachten Sie folgende, harmlos aussehende PHP-Webseite:

**Beispiel:**

```
<?php

require_once 'datenbank_mit_unicode.php';

if ($_POST) {
$sql = 'SELECT * FROM seminare WHERE titel LIKE "%' . $_POST['suche']. '%"';

$statement = $db->query($sql);
$daten = $statement->fetchAll();
}

?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/
TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
<title>Seminarsuche</title>
</head>
<body>
<form action="<?php echo $_SERVER['PHP_SELF'] ?>" method="post">
Suche: <input type="text" name="suche" />
<input type="submit" value="suchen" />
</form>
<?php if (isset($daten)): ?>
<p>Suchergebnis: </p>
<ul>
<?php foreach($daten as $seminar): ?>
<li>
<?php echo $seminar['titel'] ?>:
<?php echo $seminar['preis'] ?>?
</li>
<?php endforeach; ?>
</ul>
<?php endif; ?>
</body>
</html>
```

**Listing:** sql_injection.php

Die folgende Demonstration funktioniert in dieser Form ab PHP Version 5.3. Ältere PHP-Versionen filtern standardmäßig Formulardaten automatisch, was diese Art von Angriffen erschwert. Dies kann allerdings ganz einfach abgeschaltet werden.

Wenn Sie genau wissen wollen, warum diese Einstellung sich mit PHP 5.3 geändert hat, möchte ich Sie auf die oizielle PHP-Dokumentation verweisen:
**http://www.php.net/manual/de/security.magicquotes.php**

Falls Sie auf Ihrem Rechner noch eine Version vor 5.3 installiert haben, fügen Sie als erste Zeile in das PHP-Skript folgendes ein, um die Beispiele ebenfalls durcharbeiten zu können:

***ini_set('magic_quotes_runtime', 0);***

An sich eine kleine Seite mit einem Suchformular. Wenn Sie in das Formularfeld einen Begriff eingeben, wird die Tabelle *seminare* nach passenden *titel*-Attributen durchsucht. Probieren Sie es ruhig einmal aus, es funktioniert ganz gut für eine einfache Suche.

Und jetzt sage ich Ihnen: wenn Sie diese PHP-Webseite so auf Ihren Webspace hochladen, kann ich jede beliebige SQL-Anweisung auf Ihrer Datenbank ausführen. Ich, und zehntausende andere Programmierer mit Grundkenntnissen im Thema Websicherheit können also Datensätze ändern, löschen oder gleich die ganze Datenbank ***DROP***pen.

Sind Sie gerade am überlegen, wo Sie derartige Skripte auf Ihrem Webspace liegen haben? Oder glauben Sie mir nicht? Gut – treten wir einfach den Beweis an!

Geben Sie folgenden Text exakt so in das Suchfeld ein und senden das Formular ab (am Ende stehen zwei Minuszeichen, diese sind wichtig):

***test"; UPDATE benutzer SET passwort="12345"; --***

Schon haben alle *benutzer* als *passwort* den Wert *12345*!

Es geht noch besser! Tragen Sie folgenden Text exakt so in das Suchfeld ein:

***test"; UPDATE seminare SET preis=0.01; --***

Suchen Sie nun nach Flash-Seminaren und freuen Sie sich, wie billig diese auf einmal sind!
Ich denke, Sie sind nun genug erschrocken. Lassen Sie mich erklären, wie so etwas möglich ist.

Der böse Benutzer, also ich, hat geschickt eine zweite Abfrage in das SQL eingebaut. Da SQL mehrere Anweisungen auf einer Zeile zulässt, solange diese durch ein Semikolon getrennt sind, ist das an sich kein Problem. Das Ende des Original-SQL wurde durch das doppelte Minus (das SQL-Kommentarzeichen) auskommentiert und damit deaktiviert.

Statt einer werden nun also zwei SQL-Anfragen ausgeführt, wobei die zweite quasi beliebigen Code enthalten kann. So sind das Verändern von Admin-Passwörtern, das Anlegen von neuen Benutzern oder das Löschen bestimmter ungeliebter Datensätze prinzipiell kein Problem. Mit etwas Übung (und einem unvorsichtigen Admin) können Sie auf diese Weise sogar das Passwort des MySQL-Root-Users verändern und die Kontrolle über das komplette DBMS übernehmen.

Die einzige Möglichkeit dieses Problem zu vermeiden, ist die Benutzereingaben zu überprüfen und gegebenenfalls zu korrigieren. Hier gibt es viele Möglichkeiten, eine recht einfache Methode ist es, jede Benutzereingabe mit der PHP-Funktion ***addslashes()*** zu maskieren. Jedem potentiell

gefährlichen Zeichen, insbesondere den Anführungszeichen, wird so automatisch ein **Backslash (\)** vorangestellt.

Natürlich müssen Sie die Funktion bei jeder Benutzereingabe in Ihrem SQL einbauen. Wenn Sie *addslashes()* an einer einzigen Stelle vergessen, ist Ihr Code angreifbar. Gerade an diesem Problem kranken viele ältere PHP-Projekte, die sich um derartige Probleme erst seit relativ kurzer Zeit kümmern.

## PDO und SQL-Injections

Jetzt kommt die gute Nachricht: So lange Sie **prepared statements** mit Platzhaltern verwenden, müssen Sie sich um **SQL-Injections** keine Sorgen machen. Sämtliche potentiell gefährlichen Anweisungen in Platzhaltern werden automatisch maskiert und sind somit ungefährlich.

Daher sollten Sie immer **prepared statements** einsetzen, anstatt PHP-Variablen in Ihr SQL einbauen. So ist Ihre Webanwendung von dieser Seite aus nicht angreifbar und der Aufwand für diesen erheblichen Sicherheitsgewinn ist minimal.

Prepared statements schützen Ihre Anwendung gegen SQL-Injections - und nur dagegen!
Sie müssen auch weiterhin Schutzmaßnahmen gegen die anderen bekannten Angriffe auf Webanwendungen verwenden, so z.B. ***htmlspecialchars()*** oder ***strip_tags()*** gegen HTML und JavaScript in Benutzereingaben.

## Aufgaben zur Selbstkontrolle:

### Aufgabe 1:
Schreiben Sie das Skript *eintragen.php* aus Lektion »Einführung in PDO« um, so dass, wo sinnvoll, statt ***PDO::query()*** prepared statements verwendet werden.

### Aufgabe 2:
Schreiben Sie ein neues Skript ***regisseur_neu.php***, das ein HTML-Formular anzeigt, mit dem ein neuer Regisseur angelegt werden kann. Dieses Formular wird an eine zweite Datei ***regisseur_eintragen.php*** versendet.

### Aufgabe 3:
Das Skript ***regisseur_eintragen.php*** verwendet die Daten des Formulars aus Übung 2 und trägt einen neuen Datensatz in die Tabelle *regisseure* ein. Verwenden Sie hierfür prepared statements, um die Benutzereingaben sicher und den Code lesbar zu halten.