**Geht doch auch sicher! PHP5 mit MySQLi**

Im Sommer 2004 wurde die Version 5.0 von PHP veröffentlich und ab dann stets weiterentwickelt.
Die meisten Updates/Versionssprünge haben mit der Behebung Sicherheits- und Stabilitätsproblemen
zu tun: So sorgt eine aktuelle PHP-Version für weniger Angriffsmöglichkeiten!

Auch wenn es lange gedauert hat: Heute ist PHP5 bei den meisten Providern verfügbar
und daher für eigene Projekte nutzbar.

**MySQLi: objektorientiert und sicher**

Eine der wichtigsten Errungenschaften ist *MySQLi*.
Das i in MySQLi steht für *improved*, was übersetzt *verbessert* bedeutet.
Dass die neue MySQLi Erweiterung tatsächlich diverse Vorteile mit sich bringt sollte
nach diesem Kurs offensichtlich werden.

Die MySQL Erweiterung von PHP war bisher ein Sammelsurium loser Funktionen, die nur eine
prozedurale Programmierung ermöglichte. Dies zwang Programmierer dazu sich eigene Klassen
für das DB Handling zu schreiben. Durch PHP 5 ist es nun aber möglich auch objektorientiert
mit der DB zu kommunizieren, weil die MySQLi Erweiterung als Klasse (oder besser gesagt
als Klassen-Paket) daher kommt und einheitliche Voraussetzungen für alle schafft.

Um den Umstieg von MySQL nach MySQLi zu erleichtern, ist es aber immer noch möglich
prozedural auf die Erweiterung zuzugreifen. Das widerspricht allerdings etwas dem
"PHP 5 Spirit" der, nicht zuletzt durch die SPL und MySQLi, immer mehr in Richtung OOP geht.

**MySQL vs MySQLi**

Werfen wir mal einen Blick auf einige **Vorteile von MySQLi** gegenüber MySQL.

* Der OOP Zugriff ist einheitlich, wodurch arbeiten im Team oder an Fremdprojekten vereinfacht wird
* Größere Geschwindigkeit - Laut Hersteller kann ein vielfaches an Geschwindigkeit erreicht werden.
* Verbesserte Sicherheit - Für die PHP-MySQL Verbindung wird eine SSH-ähnliche Authentifizierung benutzt
* Das neue Binärprotokoll von MySQL wird verwendet. Dadurch stehen mehr Funktionen zur Verfügung und die Verarbeitung ist schneller und effizienter
* Keine Standardverbindung und Prepared Statements erzwingen eine saubere und somit sicherere Programmierung
* Verbesserte Trace- und Debug-Funktionen
* Multi-Queries werden unterstützt

**Prozedural oder objektorientiert?**

MySQLi wurde entwickelt damit es sich in das objektorientierte Umfeld von PHP 5 einreiht. Die prozeduralen Funktionen wurden aus Gründen der Kompatibilität und eines einfachen Umstiegs für notorische Fortschrittsverweigerer beibehalten. Deswegen ist der OOP Weg dem prozeduralen Weg klar vorzuziehen.

Bevor wir endlich zum Code-Teil kommen, schauen wir uns kurz die Unterschiede zwischen MySQL, MySQLi prozedural und MySQLi objektorientiert an.

**Für Umsteiger von MySQL**
Der Umstieg ist denkbar einfach, denn alles das sich mehr oder weniger ändert ist, dass aus dem Präfix mysql\_ ein mysqli\_ wird. Hier einige Beispiele:

// Links der MySQL Befehl, rechts das MySQLi Gegenstueck. Der Pfeil bedeutet "wird zu"

mysql\_connect() -> mysqli\_connect()

mysql\_query() -> mysqli\_query()

mysql\_fetch\_array() -> mysqli\_fetch\_array()

// und so weiter

Die Methoden beim OOP sind (meistens) ebenso eindeutig. So wird aus dem Präfix mysqli\_ der Objektname mit dem -> Operator. Beispiel:

// Angenommen unser Objekt traegt den Name $objekt dann sehen die Methoden wie folgt aus

mysqli\_query() -> $objekt->query()

mysqli\_fetch\_array() -> $objekt->fetch\_array()

// und so weiter

Nichts desto Trotz gibt es aber einige Besonderheiten bzw Unterschiede bei der prozeduralen und OOP Schreibweise. Einige prozedurale Funktionen haben nämlich keine OOP Methode, sondern stehen beim OOP als Eigenschaft zu Verfügung! Schauen wir uns das mal an...

**MySQLi Unterschiede der prozeduralen und OOP Schreibweise**
Funktionen die einen Zähl-Wert zurück liefern stehen beim OOP als Eigenschaft zur Verfügung. Das sieht wie folgt aus:

// Liefert die Anzahl der gefundenen Datensaetze

mysqli\_num\_rows() -> $objekt->num\_rows

// Liefert die Anzahl der betroffenen Datensaetze

mysqli\_affected\_rows() -> $objekt->affected\_rows

Dem affected\_rows fällt eine besondere Rolle zu. So kann man es bei MySQLi jetzt auch als Ersatz für num\_rows verwenden. Benutzt man in einem Query INSERT, UPDATE, REPLACE oder DELETE, liefert affected\_rows die Anzahl der betroffenen Datensätze. Benutzt man in einem Query SELECT, liefert affected\_rows die Anzahl der gefundenen Datensätze.

Insgesamt sind die Funktions-/Methodennamen bei MySQLi so gewählt, dass jeder halbwegs erfahrene PHP Programmierer problemlos umsteigen kann, da sich die Namen an die altbekannten MySQL Funktionen anlehnen. Es kamen aber auch neue Funktionen hinzu, von denen wir uns im weiteren Verlauf des Kapitels einige anschauen wollen.

**Datenbankverbindung öffnen/schließen**

Ohne eine Verbindung zur Datenbank wird es schwer Daten auszulesen bzw. einzutragen. Deswegen kümmern wir uns erst einmal um einen Connect.

<?php

// Neues Datenbank-Objekt erzeugen

$db = @new mysqli( 'localhost', 'root', '', 'tutorials' );

// Pruefen ob die Datenbankverbindung hergestellt werden konnte

if (mysqli\_connect\_errno() == 0)

{

 /\*

 \* Hier wird Code ausgefuehrt, wenn die Datenbankverbindung

 \* fehlerfrei hergestellt werden konnte.

 \*/

}

else

{

 // Es konnte keine Datenbankverbindung aufgebaut werden

 echo 'Die Datenbank konnte nicht erreicht werden. Folgender Fehler trat auf: <span class="hinweis">' .mysqli\_connect\_errno(). ' : ' .mysqli\_connect\_error(). '</span>';

}

// Datenbankverbindung schliessen

$db->close();

?>

Es ist nicht schwer zu erkennen was hier geschieht. Wie erzeugen ein Datenbankobjekt, wodurch automatisch eine Verbindung zur DB geöffnet wird, und prüfen ob dies fehlerfrei geschehen ist. Falls nicht, lassen wir uns den Fehlercode mit Fehlermeldung ausgeben. Am Ende des Script schließen wir die Verbindung wieder.

Aber der Reihe nach. Direkt in der ersten Zeile kommt schon etwas neues.

$db = @new mysqli( 'localhost', 'root', '', 'tutorials' );

Das "new mysqli()" entspricht dem "mysqli\_connect()" der prozeduralen Schreibweise. Dieser Connect erwartet 3 Pflicht-Parameter, nämlich Serveradresse, Benutzername, Benutzerpasswort. Optional kann man aber auch noch direkt den Datenbankname angeben! Es ist also nicht nötig erst ein mysqli\_select\_db() auszuführen. Weitere optionale Parameter beim Connect sind Port und Sockel.

Mit mysqli\_connect\_errno() prüfen wir ob ein Fehler auftrat. Der Fehlercode 0 (Null) bedeutet das alles okay ist. Falls ein Fehler aufgetreten ist lassen wir uns diesen, zusammen mit dem Fehlercode, im else-Block ausgeben.
MySQLi kommt auch mit einer eigenen Exception Klasse (mysqli\_sql\_exception), die man natürlich erweitern kann um entsprechend auf Fehler zu reagieren.

Jetzt da wir wissen wie man eine Verbindung herstellt, schauen wir uns ein Beispiel an wie man Daten ausliest.

**Datensätze auslesen (Query Methode)**

Beim auslesen von Datensätze hat sich nicht viel geändert; zumindest nicht, wenn man den gewohnten Weg geht. Hier das Script mit dem wir unsere Mod-Liste auslesen können.

<?php

// Neues Datenbank-Objekt erzeugen

$db = @new mysqli( 'localhost', 'root', '', 'tutorials' );

// Pruefen ob die Datenbankverbindung hergestellt werden konnte

if (mysqli\_connect\_errno() == 0)

{

 // Query vorbereiten und an die DB schicken

 $sql = 'SELECT `name`, `bereich` FROM `moderatoren`';

 $ergebnis = $db->query( $sql );

 // Anzahl gefunde Datensaetze ausgeben

 echo "<p>Es wurden " .$ergebnis->num\_rows. " Eintr&auml;ge gefunden.</p>";

 // Ergebnisse ausgeben

 while ($zeile = $ergebnis->fetch\_object())

 {

 echo $zeile->name. " ist zust&auml;ndig f&uuml;r " .$zeile->bereich. "<br />";

 }

 // Resourcen freigeben

 $ergebnis->close();

}

else

{

 // Es konnte keine Datenbankverbindung aufgebaut werden

 echo 'Die Datenbank konnte nicht erreicht werden. Folgender Fehler trat auf: <span class="hinweis">' .mysqli\_connect\_errno(). ' : ' .mysqli\_connect\_error(). '</span>';

}

// Datenbankverbindung schliessen

$db->close();

?>

Zuerst wird unsere Statement formuliert, das wir in $sql ablegen. In der Zeile ...

$ergebnis = $db->query( $sql );

... senden wir die Anfrage (Query) mit dem Statement ($sql) an die Datenbank und erhalten ein Ergebnis zurück das wir in $ergebnis ablegen, das dadurch zu einem Objekt wird. In der nächsten Zeile geben wir aus wie viele Datensätze gefunden wurden. Die Anzahl steckt, wie oben erwähnt, in der Eigenschaft num\_rows unseres $ergebnis Objekts.

Die eigentlichen Datensätze werden in der while-Schleife ausgegeben. Dadurch das wir die Treffer-Zeilen als Objekt fetchen, erhalten wir Zugriff auf die Felder in Form von Objekt Eigenschaften. Selbstverständlich könnten wir das Ergebnis auch mit $ergebnis->fetch\_array() ausgeben. In dem Fall würden wir auf die Zeilen als Array zugreifen: $zeile['name'] und $zeile['bereich']

Jetzt schauen wir uns einen alternativen MySQLi Weg an, wie wir an die Ergebnisse kommen können.

**Datensätze auslesen (Prepare Methode)**

Neu in MySQLi ist, dass man Anweisungen an die Datenbank vorbereiten kann. Dadurch teilt man der DB mit welche Daten gesendet werden und wie damit zu verfahren ist. Es wird also nicht direkt eine Query an die DB geschickt, sondern erst eine Art Vorbote mit Informationen über die eigentliche Anweisung. Die Anweisung kann man zu einem beliebigen Zeitpunkt, beliebig oft an die DB senden.
Man bereitet also eine Anweisung ein mal vor und kann diese dann beliebig oft senden - das ist, besonders beim Eintragen in die DB, wesentlich schneller, übersichtlicher, flexibler und sicherer! Zunächst schauen wir uns aber an wie man Prepared Statements (Vorbereitete Anweisung) beim Auslesen verwendet.

<?php

// Neues Datenbank-Objekt erzeugen

$db = @new mysqli( 'localhost', 'root', '', 'tutorials' );

// Pruefen ob die Datenbankverbindung hergestellt werden konnte

if (mysqli\_connect\_errno() == 0)

{

 $sql = 'SELECT `name`, `bereich` FROM `moderatoren` ORDER BY `name`';

 // Statement vorbereiten

 $ergebnis = $db->prepare( $sql );

 // an die DB schicken

 $ergebnis->execute();

 // Ergebnis an Variablen binden

 $ergebnis->bind\_result( $name, $bereich );

 // Ergebnisse ausgeben

 while ($ergebnis->fetch())

 {

 echo $name. " ist zust&auml;ndig f&uuml;r " .$bereich. "<br />";

 }

}

else

{

 // Es konnte keine Datenbankverbindung aufgebaut werden

 echo 'Die Datenbank konnte nicht erreicht werden. Folgender Fehler trat auf: <span class="hinweis">' .mysqli\_connect\_errno(). ' : ' .mysqli\_connect\_error(). '</span>';

}

// Datenbankverbindung schliessen

$db->close();

?>

Das Zusammenstellen der Abfrage ist identisch zum vorherigen Beispiel. Diesmal wird $sql allerdings nicht direkt per $ergebnis = $db->query( $sql ); an die DB gesendet, sondern wir teilen der DB mit einem ...

$ergebnis = $db->prepare( $sql );

... mit was wir von der Abfrage erwarten. Wir bereiten (prepare) also die eigentliche Anweisung vor. Ausgeführt wird die Anweisung dann mit ...

$ergebnis->execute();

Dieses execute() kann man unendlich oft an die DB senden und es wird immer die zuletzt festgelegte Anweisung ausgeführt. Das hört sich momentan nach "na und?" an, ist aber beim befüllen der DB von großem Vorteil, wie wir weiter unten noch sehen werden. Das Abfrageergebnis steht jetzt zur Verfügung. Statt es direkt in einer Schleife auszugeben, binden wir die Felder der einzelnen Zeilen unseres Ergebnis an selbst definierte Variablen.

$ergebnis->bind\_result( $name, $bereich );

Die Anzahl der Parameter (Variablen) muß identisch mit der Anzahl der Felder unserer Abfrage sein. Ausgegeben wird das Ergebnis dann in einer while-Schleife ...

while ($ergebnis->fetch())

{

 echo $name. " ist zust&auml;ndig f&uuml;r " .$bereich. "<br />";

}

... bei der mittels $ergebnis->fetch() Zeile für Zeile auf unser Ergebnis zugegriffen wird. Die Variablen $name und $bereich werden bei jedem Durchlauf mit den neuen Werten gefüllt.
Diese Art des Ergebnis binden kann wesentlich übersichtlicher sein als ein Zugriff auf Objekt Eigenschaften oder assoziative Arrays. Desweiteren kann man sich auf diese Art teilweise das Hantieren mit Aliases in der DB sparen.

Man kann aber nicht nur Abfrageergebnisse an Variablen binden, sondern auch Variablen an Anweisungen. Das, zusammen mit der Fähigkeit Anweisungen vorzubereiten, ermöglicht es uns sehr komfortabel und sicher Daten in die DB zu schreiben.

**Datensätze schreiben mit Prepared Statements**

Zuerst basteln wir uns ein ganz schlichtes Formular, damit wir auch etwas zum schreiben in die Datenbank bekommen.

<form method="post" action="mysqli\_schreiben.php">

Name: <input type="text" name="name" /><br />

Bereich <input type="text" name="bereich" /><br />

<input type="submit" name="submit" value="Eintragen" />

</form>

Zur Demonstration reichen die beiden Felder aus. Die übermittelten Daten werden von unserem Script entgegengenommen und verarbeitet. Werfen wir zuerst einen Blick auf das vollständige Script und gehen dann die neuen Zeilen durch.

<?php

if (isset( $\_POST['submit'] ))

{

 // Maskierende Slashes aus POST Array entfernen

 if (get\_magic\_quotes\_gpc()) { $\_POST = array\_map( 'stripslashes', $\_POST ); }

 $name = $\_POST['name'];

 $bereich = $\_POST['bereich'];

 // Neues Datenbank-Objekt erzeugen

 $db = @new mysqli( 'localhost', 'root', '', 'tutorials' );

 // Pruefen ob die Datenbankverbindung hergestellt werden konnte

 if (mysqli\_connect\_errno() == 0)

 {

 $sql = 'INSERT INTO `moderatoren` (`name`, `bereich`) VALUES (?, ?)';

 $eintrag = $db->prepare( $sql );

 $eintrag->bind\_param( 'ss', $name, $bereich );

 $eintrag->execute();

 // Pruefen ob der Eintrag efolgreich war

 if ($eintrag->affected\_rows == 1)

 {

 echo 'Der neue Eintrage wurde hinzugef&uuml;gt.';

 }

 else

 {

 echo 'Der Eintrag konnte nicht hinzugef&uuml;gt werden.';

 }

 }

 else

 {

 // Es konnte keine Datenbankverbindung aufgebaut werden echo 'Die Datenbank konnte nicht erreicht werden. Folgender Fehler trat auf: <span class="hinweis">' .mysqli\_connect\_errno(). ' : ' .mysqli\_connect\_error(). '</span>'; } // Datenbankverbindung schliessen $db->close(); } ?>

Der obere Teil sollte klar sein. Erst wird geprüft ob das Formular abgeschickt wurde, dann werden maskierende Slashes entfernt, die Variablen umgeladen, eine DB-Verbindung hergestellt und dann geht es los.

Wir stellen wieder unsere Anweisung zusammen und legen diese in $sql ab. Aber diesmal sehen wir sofort etwas besonderes.

$sql = 'INSERT INTO `moderatoren` (`name`, `bereich`) VALUES (?, ?)';

Im VALUES() Abschnitt werden keine Variablen eingegeben, sondern wir setzen hier Platzhalter ein. Für jedes Feld das wir hinzufügen möchten (hier name und bereich) muss ein Value respektive Platzhalter angegeben werden.

Der Platzhalter muß ein Fragezeichen sein und darf nicht von Quotes oder Double Quotes eingeschlossen sein!

Anschließend senden wir das vorbereitete Statement an die DB, damit diese weiß was zu tun ist. Nun kommen wir zum Kern des Ganzen:

$eintrag->bind\_param( 'ss', $name, $bereich );

Hier binden wir Variablen an das Statement das wir zuvor an die DB geschickt haben. Der erste Parameter, hier 'ss', teilt der Datenbank mit um welche Daten es sich handeln wird, die in die DB geschrieben werden sollen. Anhand dessen wird die DB eigenständig das escapen (unschädlich machen von Benutzereingaben) übernehmen - dieser Parameter stellt also eine Typbezeichnung dar.

**Es gibt 4 Typen:**

* s = String
* i = Integer (Ganzzahl)
* d = Double (Gleitkommazahl)
* b = BLOB (Binary Large Object - also eine binäre Datei)

Die weiteren Parameter sind die eigentlichen Daten, die in die DB geschrieben werden sollen.

Wichtig: Jeder Platzhalter im Prepared Statement muss einen korrespondierenden Wert in bind\_param() haben.
Jeder Wert in bind\_param() muss eine Typbestimmung haben!
Die Anzahl von Platzhalter, Typ-Bezeichner und Werte muss identisch sein, sonst kommt es zu Fehler!

**Ergänzendes Beispiel:**

$sql = 'INSERT INTO `tabelle` (`id`, `kategorie`, `produktname`, `preis`) VALUES (?, ?, ?, ?)';

// uebergeben wird: Integer, String, String, Double

$eintrag->bind\_param( 'issd', 27, 'Werkzeug', 'Schraubenzieher', '39.99' );

Der Rest des Scripts ist selbsterklärend. Das execute() haben wir schon kennengelernt.
Durch die Eigenschaft $eintrag->affected\_rows ermitteln wir, ob der Eintrag erfolgreich war.

**Hinweis zu bind\_param() und execute()**
Wenn einmal Variablen an ein Prepared Statement gebunden sind, können diese Variablen beliebig im Script geändert und anschließend mit execute() an die DB geschickt werden. Um mehrere Einträge zu schreiben ändert man also nur den Wert der Variablen und schickt diese an die DB - dies kann z.B. in einer Schleife geschehen. Beispiel:

$sql = 'INSERT INTO `tabelle` (`vorname`, `nachname`) VALUES (?, ?)';

$eintrag = $db->prepare( $sql );

// Variablen an die Anweisung binden

$eintrag->bind\_param( 'ss', $vorname, $nachname );

// Erster Eintrag

$vorname = 'Max';

$nachname = 'Mustermann';

$eintrag->execute();

// Zweiter Eintrag

$vorname = 'Fritz';

$nachname = 'Fischer';

$eintrag->execute();

// Dritter Eintrag

$vorname = 'Susi';

$nachname = 'Sauer';

$eintrag->execute();

// usw

Quellen: (Stand: 02/2012)

* [http://www.phpbuddy.eu](http://www.phpbuddy.eu/mysqli-erweiterung-php-5.html)
* [http://www.php-homepage.de](http://www.php-homepage.de/manual/function.mysql-real-escape-string.php)
* [http://www.php-lernen.org](http://www.php-lernen.org/php-mysql-sicherheit-sql-injections-mysql_real_escape_string.html)