Kapitel 5

PHP

5.1 Wiederholung

Bevor mit dem DB-Zugriff begonnen wird, sollen Sie nochmals folgende Themen, die im Vorjahr besprochen worden sind, durcharbeiten:

- Variablen
- Arrays
- Kontrollstrukturen (Schleifen, Verzweigungen)
- Funktions-, Klassendefinition
- Dateizugriffe
- Cookies und Sessions
- Zugriff auf GET- bzw. POST-Variablen.

5.2 Datenbankanbindung

Der Zugriff aus PHP auf Datenbanken kann auf verschiedenste Weisen erfolgen, die in den folgenden Kapiteln aufgezeigt werden sollen.

5.3 Deprecated: mysql-Funktionen

Um mit einer MySQL-Datenbank zu kommunizieren, gibt es bei der alten Erweiterung nur eine Ansammlung von Funktionen. Funktionalitäten wie zum Beispiel Prepared Statements kann man nur über den Umweg mehrerer Abfragen gehen. Und an eine reine objektorientierte Programmierung ist schon mal gar nicht zu denken. Zum Zugriff muss zuerst eine Verbindung (`mysql_connect`) hergestellt werden. Dazu ist der Host, auf dem der MySQL-Server läuft, der Benutzername sowie das Kennwort anzugeben. Anschließend muss noch eine Datenbank auf dem DBSY ausgewählt werden (`mysql_select_db`).

Danach können die SQL-Anweisungen folgen (`mysql_query`). Im Falle einer SQL-Abfrage liefert diese Funktion alle Tupel zurück, welche der Reihe nach mit `mysql_fetch_array` abgearbeitet werden können.

Die Funktionen sind seit PHP 5.5 als "deprecated" (veraltet) eingestuft und seit PHP7 nicht mehr verfügbar.

5.4 mysqli - Methoden

mysqli dagegen kann man sowohl prozedural als auch objektorientiert nutzen. Im Gegensatz zu der alten MySQL-Extension bietet die neue Variante zahlreiche verbesserte Funktionen (das ‚i' steht für improved). Dazu zählen vor allem die Unterstützung von Prepared Statements, Transaktionshandling und zahlreiche Verbesserungen im Kern der API, welche die Performance drastisch erhöhen können (um bis zu 40%). Und eben genau diese Prepared Statements verschaffen der Webanwendung wesentlich mehr Sicherheit und Performance.

5.4.1 Verbindungsaufbau

Die MySQLi-Klasse verfügt über einen Konstruktor, mit dem die Verbindung zum MySQL-Server aufgebaut wird. Man muss einfach nur die Zugangsdaten angeben.

```
$mysqli = new mysqli('host', 'user', 'passwort', 'datenbank');
```

und hat dann Zugriff auf die Eigenschaften und Methoden.

```
echo $mysqli -> server_info;
```

5.4.2 init()

Sollen für die Verbindung stattdessen erweiterte Parameter, beispielsweise für eine Verschlüsselung der Client-Server-Kommunikation, gesetzt werden, so ist mit der Funktion mysqli_init() zunächst ein MySQLi-Objekt zu erzeugen. Über die entsprechenden Methoden, beispielsweise ssl_set(), können dann die entsprechenden Einstellungen vorgenommen werden, bevor die eigentliche Verbindung mit der Methode real_connect() hergestellt wird. Diese Methode akzeptiert dieselben Parameter wie der Konstruktor.

```
$mysqli = mysqli_init();
$mysqli -> options (MYSQLI_INIT_COMMAND, 'SET NAMES \'utf8\'');
$mysqli -> real_connect('host', 'user', 'pw', 'db');
```

5.4.3 close()

Trennt eine bestehende Verbindung.

```
$mysqli = new mysqli('host', 'user', 'pw', 'db');
$mysqli -> close();
```

5.4.4 query() - non prepared statements

Die Methode `query()` der Klasse MySQLi liefert ein Objekt der Klasse MySQLi_Result. Dieses kapselt die Ergebniszeilen der Anfrage. Da die Klasse MySQLi_Result leider nicht die Schnittstelle Iterator anbietet, kann das Objekt nicht direkt mit dem foreach-Operator verwendet werden. Stattdessen ist eine entsprechende while-Schleife zu verwenden:

```
$mysqli = new mysqli('host', 'user', 'pw', 'db');

if (!$mysqli->query("DROP TABLE IF EXISTS TAB1") ||
!$mysqli->query("CREATE TABLE TAB1(id INT)") ||
!$mysqli->query("INSERT INTO TAB1(id) VALUES (1)") ||
!$mysqli->query("INSERT INTO TAB1(id) VALUES (2)") ||
!$mysqli->query("INSERT INTO TAB1(id) VALUES (3)")) {
  echo "Table creation failed: (" . $mysqli->errno . ") " . $mysqli->error;
}
```

```
$query  = 'SELECT id FROM tab1 WHERE id > 1';
$result = $mysqli->query($query);
```

In der Variable $result befindet sich nun die Ressourcenkennung/Referenz auf die Abfrage. Und die benötigt man, um sich in einer Schleife die Daten zu holen.

```
while ($row = $result->fetch_assoc()){
  //...
}
```

5.4.5 Prepared Statements

Prepared Statements sind vorbereitete Datenbank-Queries ohne Werte für die einzelnen Parameter. Statt der wirklichen Parameterwerte werden in der Anweisung Variablen verwendet, die dann zur Laufzeit befüllt werden. Ein Prepared Statement ist ein Anweisungstemplate. So ein Template wird beim Aufruf der prepare()-Methode analysiert, kompiliert und optimiert. Ab dann wird bei wiederholten Anweisungen immer dieses vorkompilierte und optimierte Template mit den geänderten Werten benutzt. Dadurch verbrauchen Prepared Statements wesentlich weniger Ressourcen und sind schneller als herkömmliche Standard-Queries. Mit Prepared Statements sind normalerweise keine SQL Injections möglich. Die Variablen müssen nicht escaped werden, die Datenbank prüft dies selbst.

Die Arbeit mit Prepared Statements läuft folgendermaßen ab:

- Vorbereiten
- Parameter setzen
- Ausführen
- Daten holen

5.4.5.1 Vorbereiten

Dazu benötigt man ein SQL-Statement, gegebenenfalls mit so genannten Wildcards. Üblich ist das Fragezeichen. Das Statement selber wird dann mit prepare eingeleitet.

```
$mysqli = new mysqli('host', 'user', 'pw', 'db');
$query  = 'INSERT INTO TAB1(id) VALUES (?)';
if (!($stmt = $mysqli->prepare($query))) {
  echo "Prepare failed: (" . $mysqli->errno . ") " . $mysqli->error;
}
```

5.4.5.2 Parameter setzen

Bei obiger Anweisung wollen wir einen Wert in eine Tabelle schreiben und dazu müssen wir das Fragezeichen einen Wert zuweisen. Dies erfolgt mit der Methode bind_param.

```
if (!$stmt->bind_param("i", $id)) {
  echo "Binding parameters failed: (" . $stmt->errno . ") " . $stmt->error;
}
```

Beim ersten Parameter definiert man den Typ der einzelnen Werte. Es gibt dabei vier Möglichkeiten:

- i Integer (Ganzzahl)
- d Double (Fließkommazahl)
- s String (Zeichenkette)
- b Blob

5.4.5.3 Ausführen

```
if (!$stmt->execute()) {
  echo "Execute failed: (" . $stmt->errno . ") " . $stmt->error;
}
```

5.4.5.4 Daten holen

Dies erfolgt über die Methode get_result().

```
/* Fehlerbehandlung bewusst weggelassen */
$mysqli = new mysqli('host', 'user', 'pw', 'db');
$query  = 'SELECT id FROM tab1 WHERE id > ?';
$stmt   = $mysqli->prepare($query);
$id     = 0;
$stmt  -> bind_param('i', $id);
$stmt  -> execute();
$result = $stmt -> get_result();
while ($row = $result -> fetch_assoc()) {
  print_r($row);
}
```

5.4.6 Fehlerbehandlung

Hier reagiert die MySQLi-Erweiterung genau so wie die von MySQL. Wenn die Verbindung zum Server nicht gelingt, so wird ein Warning ausgeworfen. Wenn man dagegen diese Meldung unterdrückt hat, so gibt es die beiden Eigenschaften/Funktionen connect_errno und connect_error.

```
@$mysqli = new mysqli('host', 'user', 'falsches_pw', 'db');
if ($mysqli->connect_errno) {
  echo 'Verbindung fehlgeschlagen. Wegen '.$mysqli -> connect_error;
}
```

Im Gegensatz zur MySQL-Erweiterung gibt es einen großen Unterschied. Bei MySQLi gibt es eine Trennung zwischen Verbindungs- und sonstigen Fehlern. Ist also die Syntax fehlerhaft oder die Abfrage an sich falsch, so muss man dafür auf error beziehungsweise errno zugreifen:

```
$mysqli = new mysqli('host', 'user', 'pw', 'db');
$query  = 'SELECT bla FROM blubb WHERE bla=\'blubber';
echo $mysqli->errno.'-'.$mysqli->error;
```

Um Fehler abzufangen, gibt es die üblichen Bordmittel von PHP. Entweder mit reinen if-Bedingungen oder mit einem try ... catch Block. Denn wenn eine Abfrage einen Fehler aufweist, so gibt query() immer ein false zurück.

```
$mysqli = new mysqli('host', 'user', 'pw', 'db');
try {   // Tabelle bla existiert nicht
  $query  = 'SELECT * FROM bla';
  if(!$mysqli -> query($query)) {
    throw new Exception($mysqli -> error);
  }
} catch (Exception $e) {
  echo $e -> getMessage();
}
```

5.4.7 Transaktionen

```
//Autocommit deaktivieren
//OOP
$mysqli = new mysqli('host', 'user', 'pw', 'db');
$mysqli->autocommit(FALSE);

$mysqli->begin_transaction();
$bla   = $mysqli -> query('INSERT INTO bla ...)');
$blubb = $mysqli -> query('INSERT INTO blubb ...)');
if ($bla && $blubb) {
  $mysqli->commit();
} else {
  $mysqli->rollback();
}
```

5.5 PDO - 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. Zusammengefasst ist PDO eine abstrahierte Datenbankschnittstelle, die mittels verschiedener Treiber (MySQL, PostgreSQL, Oracle,...) auf verschiedene Datenbanken zugreifen kann.

Um zu überprüfen, welche Datenbanktreiber für die Verwendung mit PDO auf ihrem System bereitstehen, können sie folgenden PHP-Auszug verwenden:

```
echo "<pre>";
  print_r(PDO::getAvailableDrivers());
echo "</pre>";
```

PDO stellt drei Klassen zur Verfügung:

- `PDO`: Basisklasse; repräsentiert die Verbindung zwischen PHP und einem Datenbankserver
- `PDOStatement`: Klasse, die Prepared Statements und nach Ausführung des Statements den zurückgegebenen Ergebnissatz repräsentiert.
- `PDOException`: Klasse, die einen von PDO ausgelösten Fehler darstellt.

5.5.1 Verbindungsauf- und abbau zu einer DB

Die Verbindung zu einer Datenbank wird für alle möglichen Datenbanktypen durch das Erstellen eines Objekts der Klasse PDO erzeugt. Der Konstruktor der Klasse erwartet Angaben zu Datenbankserver und Datenbankname, sowie, optional, Benutzername und Kennwort.

```
try{
  $handle=new PDO("mysql:host=127.0.0.1;dbname=datenbank;port=3306",'username2','geheim');
  echo "Verbindung erfolgreich";
}catch(PDOException $ex){
  //Fehlerbehandlung
  echo $ex->getMessage();
}
unset($handle);
```

5.5.2 init - Optionen für den Verbindungsaufbau

In Anlehnung an Kapitel 5.4.2soll im folgenden Auszug kurz gezeigt werden, wie mit PDO Optionen eingestellt werden können:

```
$server   = 'mysql:dbname=datenbank;host=localhost; port=3306';
$user     = 'username';
$password = 'geheim';
$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);
```

5.5.3 Daten auslesen - query

Die Methode query ist ausschließlich für SELECT-Anweisungen gedacht. Für INSERT, UPDATE, REPLACE oder DELETE sollte man die schon in Kapitel vorgestellte exec-Methode nutzen. Die query-Methode gibt ein PDOStatement-Objekt zurück. Und darum existieren auch verschiedene Möglichkeiten, um die Ergebnisse zu verarbeiten.

```
$handle=new PDO("mysql:host=127.0.0.1;dbname=datenbank;port=3306",'username','geheim');
$sql='Select * from Meldung';
$ergebnis=$handle->query($sql);
echo "Anzahl der Tupel:".$ergebnis->rowCount()."<br/>";
while($row=$ergebnis->fetch()){
    print_r($row);
    echo "<br/>";
}
```

Folgende Methoden stellt das PDOStatement Objekt zur Auswertung der Ergbnisse zur Verfügung:

- `fetchAll`
- `fetch`
- `fetchObject`
- `fetchColumn`

5.5.4 Daten eintragen,.. - 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.

```
...
$query = 'DELETE FROM Meldung WHERE ID > 2';
$num   = $handle -> exec($query);
echo $num;
...
```

5.5.5 Prepared Statements in PDO

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

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

5.5.5.1 prepare

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

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

5.5.5.2 bind, execute

Die Bindung der Variablen an die Platzhalter weicht von der Verwendung von mysqli etwas ab. Die Methode `bindParam()` wird folgendermaßen verwendet:

`$prepared=bindParam(Parameter-ID,Variable [,Datentyp] [,Länge]);`

- Der erste Parameter Parameter-ID identifiziert den richtigen Platzhalter und ersetzt diesen durch die angegebene Variable. PDO bietet den Platzhalter `?` und einen Platzhalter in der Form `:name`.
- Als zweiter Parameter folgt die Angabe der Variablen, die verwendet werden soll.
- Optionale Parameter sind Datentyp und Länge. Beim Datentyp werden PDO-Konstanten in der Form
  - PDO::PARAM_BOOL (Boolsche Variable)
  - PDO::PARAM_INT (Ganzzahl)
  - PDO::PARAM_NULL (NULL-Wert von MySQL)
  - PDO::PARAM_STR (Zeichenkette); hier gibt es zusätzlich die Möglichkeit, als vierten Parameter noch die Länge vorzugeben.

```
...
$message='SQL-Injection';
$entry='Immer noch sind SQL-Injections moeglich';
$query='INSERT INTO meldung (datum,name,eintrag) VALUES (now(),:mesg,:eintrag)';
$stmt=$handle->prepare($query);
$stmt->bindParam(':mesg', $message, PDO::PARAM_STR);
$stmt->bindParam(':eintrag', $entry, PDO::PARAM_STR);
$stmt->execute();
echo "Anzahl der eingefuegten Zeilen: ".$stmt->rowCount();
...
```

5.5.6 PDO - Fehlerbehandlung - Die Exception-Klasse

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

5.5.6.1 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 eintra existiert nicht
  $query  = 'SELECT eintra FROM meldung WHERE id > 1';
  $result = $pdo -> query($query);
}
catch (PDOException $e) {
  print_r($e);
}
```

Empfehlenswert wäre es bei größeren Projekten, eine eigene Exception-Klasse einzurichten.