WORUM ES GEHT

MYSQL IN PHP NUTZEN

Du hast gesehen, wie man per MySQL Query mit einem Datenbankserver kommunizieren kann. Das Ziel ist jedoch, dass deine Applikation die benötigten Daten selbst holt und liefert. Hierfür soll die Datenverarbeitung direkt in PHP geschrieben werden. Nachfolgend erfährst du wie du das mit wenigen Zeilen Code tun kannst, und wie du auch hier dafür sorgen kannst, trotz Mitwirkungsmöglichkeiten der Benutzer (User-Input) deine Applikation sicher gestaltest.

In PHP stehen dafür zwei Erweiterungen zur Verfügung: MySQLi (MySQL Improved) oder der PDO (PHP Data Objects). Beide Methoden bieten eine Vielzahl von Funktionen zur Interaktion mit der Datenbank. Im Nachfolgenden wird die Datenbank-Kommunikation über PDO erläutert, welches rein objektorientiert ist. Das Prinzip ist bei beiden jedoch sehr ähnlich (die Aufgabe ist ja die gleiche), wenn du später MySQLi nutzen möchtest, kannst du den Code bestimmt ohne grosse Probleme darauf anpassen.

WAS IST PDO

PDO (PHP Data Objects) ist eine Erweiterung in PHP, die eine einheitliche Schnittstelle für den Zugriff auf verschiedene Datenbanken bietet. Sie bildet eine Abstraktionsschicht (Abstraction Layer), das heisst, sie ermöglicht Entwicklern, Datenbankoperationen durchzuführen, ohne sich um die spezifischen Details der verwendeten Datenbank kümmern zu müssen – hast du einmal PDO nutzen gelernt, kannst du deinen Code auch für andere Datenbanksysteme als MySQL nutzen, ohne viel anpassen (oder neu lernen) zu müssen. Sie unterstützt z. B. MySQL, PostgreSQL, SQLite. PDO bietet ausserdem eine einfache API für **Prepared Statements** und implementiert **Exceptions (Ausnahmen)**, was mehr Möglichkeiten für das Fehlerhandling bietet.

Bevor du loslegen kannst…

Um PHP mit MySQL zu verwenden, muss zunächst eine Verbindung zur MySQL-Datenbank hergestellt werden. Eine Verbindung zum Datenserver ist immer notwendig, solange du im PHPMyAdmin auf dem Localhost arbeitest, merkst du davon jedoch nichts, weil du automatisch als Root-user angemeldet bist. Den Root User kannst du auch für deine Test-Skripts benutzen, er unterscheidet sich leicht auf Windows und Mac:

OS	Username	Passwort
Windows (XAMPP)	root	*(leer)*
Mac OS (MAMP)	root	root

Die Verbindung wird über eine einzelne Zeile Code – die Instanzierung der PDO-Klasse – hergestellt und wird in dem PDO-Objekt, das zurückgegeben wird, gespeichert.

```
$pdo = new PDO('mysql:host=localhost;dbname=testdb', $username, $password);
```

Ist dein Skript erst einmal verbunden, kann das PDO-Objekt diverse Anfragen (Queries oder auch Statements) an den Datenbankserver schicken, und danach auch Daten empfangen und zurückliefern, sofern diese vom Server bereitgestellt wurden. Die empfangene Information unterscheidet sich je nach Art der Anfrage – bei READ-Statements werden Daten vom DB Server bereitgestellt, bei Datenmanipulationen wie CREATE, UPDATE und DELETE erhältst du lediglich die Information, ob die Anfrage erfolgreich war (TRUE oder FALSE).

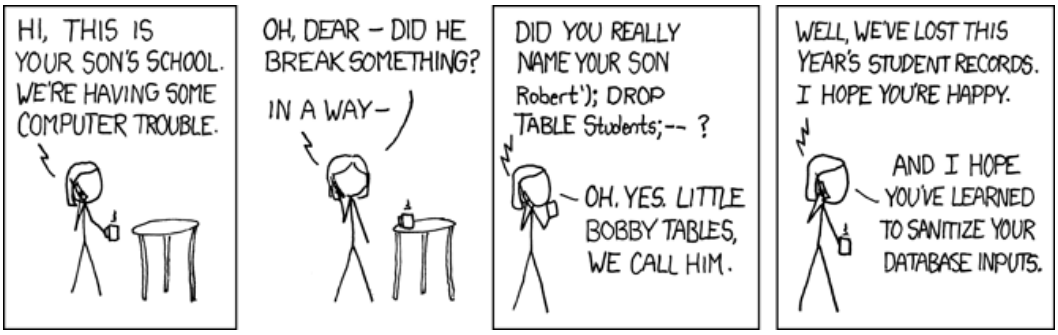
```
$stmt = $pdo->query("SELECT * FROM blogposts ORDER BY datum DESC");
$posts = $stmt->fetchAll();
```

Prepared Statements und SQL Injections – was ist denn das?

Um den Sinn von Prepared Statements zu verstehen, muss zuerst ein Blick auf das Thema Sicherheit geworfen werden. SQL Queries selbst werden zwar von PHP an den SQL-Server übermittelt, oft wird jedoch im Befehl auch User-Input eingebaut (z.B. Variablen aus POST- oder GET-Daten).

Wenn z.B. Formular-Eingaben nicht richtig überprüft werden, kann der Angreifer die Kontrolle über die Datenbank übernehmen. Das bedeutet, er könnte vertrauliche Daten stehlen, verändern oder löschen (hier wird das gut erklärt: https://www.youtube.com/watch?v=QY0TUSW3OCY)

Anders als bei Form Injections, wo mit strip_tags() gearbeitet werden kann, um ungewollten HTML Code zu entfernen, ist es hier schwieriger, Befehlsteile in User-Input zuverlässig zu finden zu entfernen, wir müssten uns da auf unser Vorstellungsvermögen verlassen, denn SQL Befehle sind einfach Strings – wie die Formulardaten.



Beliebtes comic zur Darstellung des Problems – Quelle: https://xkcd.com

Prepared Statements lösen dieses Problem, in dem bei Anfragen an den DB Server zuerst Daten und Logik getrennt werden. Das heisst, der SQL-Server erhält erst einen Befehl ohne Daten und dann erst die Daten. Der DB Server weiss so schon vor der Ausführung, WAS er ausführen wird, und lässt sich davon auch durch zusätzliche Befehle im Datenteil nicht mehr von seiner Aufgabe abbringen.

Benutzer-Query ohne Prepared Statement

Der Befehl enthält eine Variable. PHP ersetzt diese durch deren Inhalt und übermittelt diesen String inkl. Daten direkt an den Server, der diesen direkt ausführt.

```
$stmt = $pdo->query("SELECT * FROM users WHERE email = '".$_POST['email']."'");
$user = $stmt->fetch();
```

Benutzer-Query mit Prepared Statement

Beim Prepared Statement wird zuerst der Befehl geschickt (Vorbereitung). Dieser kann nun schon validiert und verstanden werden. Da wo die Variablen hinkommen, ist ein Platzhalter. Erst danach bei der Ausführung werden Daten geliefert, die dann auch nur als Daten interpretiert werden, nicht als Teil des Befehls.

```
$stmt = $pdo->prepare('SELECT * FROM users WHERE email = :email');
$stmt->execute(['email' => 'example@example.com']);
$user = $stmt->fetch();
```

Was sind Exceptions (Ausnahmen)?

In PHP sind Exceptions spezielle Ereignisse, die während der Ausführung eines Programms auftreten und den normalen Ablauf des Programms unterbrechen. Sie werden verwendet, um Fehler zu behandeln, die während der Laufzeit auftreten können, und das Programm ansonsten unterbrechen würden (Fatal Errors).

Stell dir vor, du schreibst ein Programm, das eine Datei öffnet und deren Inhalt liest. Was passiert, wenn die Datei nicht existiert? Ohne Exceptions würde das Programm einfach abstürzen (Error 500 oder eine weisse Seite). Mit Exceptions kannst du diesen Fehler aber abfangen und eine passende Reaktion darauf ausführen, zum Beispiel eine Fehlermeldung anzeigen, ein Log File schreiben oder einen anderen weg probieren.

Exceptions ermöglichen den folgenden Vorgang:

1. **Werfen (Throw)**: Wenn ein Fehler auftritt, wird eine Ausnahme “geworfen”.
2. **Fangen (Catch)**: Du kannst den Code, der potenziell Fehler verursachen könnte, in einen try-Block setzen. Wenn eine Exception geworfen wird, springt das Programm in den entsprechenden catch-Block, wo du den Fehler nach deinen Ideen behandeln kannst.

Exceptions können unter anderem auch bei SQL-Injections helfen. Diese können zwar mit Prepared Statements verhindert werden, wenn dadurch jedoch ein Fehler erzeugt wird, könnte die Fehlermeldung dem Angreifer einen Hinweis auf die Programmstruktur geben. Wenn du den Fehler abfängst, und eine generische, ungenaue Fehlermeldung ausgibst, schützt du deine Applikation noch besser. Angreifer versuchen nämlich nicht immer nur, eine Applikation direkt anzugreifen, sondern oft beginnen sie damit, durch das Provozieren von Fehlern mehr Informationen über das Setup, Versionen und Dateistruktur zu erhalten. Indem du die Informationen unbrauchbar machst, kannst du einen Angreifer also oft schon vor dem eigentlichen Angriff loswerden.

Sie enthalten ausserdem detaillierte Fehlermeldungen und Stack-Traces, die dir helfen, den genauen Ort und die Ursache eines Fehlers zu identifizieren. Dies ist besonders nützlich bei der Arbeit mit Datenbanken, wo Fehler oft schwer zu diagnostizieren sind.

AUFGABE

1. Lese den Text
2. Überlege dir eine Faustregel, wann Prepared Statements gebraucht werden und wann nicht
3. Betrachte die Unterschiede der Code-Beispiele mit und ohne Prepared Statements – welche Vorteile könnte das Beispiel MIT Prepared Statement für dich sonst noch haben?
4. Weitere Fragen? Bringe sie mit in den Unterricht!