Denne guide er oprindeligt udgivet på Eksperten.dk

COMPUTERWORLD

Prepared Statements under MySQLI - kom igang

Det gamle (og forældede) MySQL-API er stadig udbredt i skræmmende grad, selvom der findes langt sikrere alternativer. Et af dem hedder MySQLI med Prepared Statements. Denne guide er beregnet for begyndere i MySQLI - og er på ingen måde uddybende.

Skrevet den **19. jul 2012** af **olebole** I kategorien **Programmering / PHP** | ⭐⭐⭐⭐☆

Historik:
19.03-2012: Tilføjet UPDATE eksempel
19.07-2012: Opdatering om Zend's anbefalinger

Zend (som vedligeholder PHP) anbefaler nu direkte, at man fravælger MySQL-API'et til nyudvikling. I stedet anbefales MySQLI eller PDO.

MySQL
Det gamle MySQL-API, som rigtig mange stadig bruger, bygger på en højst usikker tilgang til databaseforespørgsler. Det bygger på konkatenering (sammenklistring) af strengstumper til en SQL-kommando, som efterfølgende afvikles direkte mod databasen - f.eks:

```
$user = $_POST['username'];
$sql = "SELECT `id`, `points` FROM `mytable` WHERE `user`='.$user.' LIMIT 1";
$res = mysql_query($sql);
$row = mysql_fetch_assoc($res);
echo 'Brugeren '.$user.' med id '.$row['id'].' har '.$row['points'];
```

Dette er i udgangspunktet en yderst uhensigtsmæssig fremgangsmåde, eftersom variabler - som ofte kommer udefra (altså noget, jeg som udvikler ikke selv har skrevet) - bliver sendt direkte ned i databasen og afviklet mod denne.

Man kan lappe på problemet ved at escape værdier med funktioner som mysql_real_escape_string, men det er - og vil altid være - lapperier på en i udgangspunktet forkert fremgangsmåde. Da mysql_real_escape_string kun kan bruges på strenge, står løsningen stadig åben for SQL-injection, hvis der bliver brugt tal i SQL'en.

Desuden virker mysql_real_escape_string ikke på strenge i multibyte tegnsæt - med undtagelse af utf-8. Sidst, men ikke mindst, er løsningen afhængig af, at udvikleren husker at bruge den hver eneste gang - hvilket med garanti ikke sker.

Meget ofte vil mysql_real_escape_string o.lign. 'hacks' ydermere resultere i uønskede dobbelt escapings med deraf følgende backslashes (\) i de gemte data, når disse udskrives.

Desværre ses brugen af dette API ofte i lærebøger og tutorials. Lærebøger om webkode forældes hurtigt, og tutorialforfattere har ofte lært fra ældre bøger eller andre tutorials uden at opdatere deres viden.

Endnu værre er det, at der stadig undervises i API'et på flere uddannelser på beskedent niveau. Hvad det skyldes, skal jeg ikke kunne sige, men det er i hvertfald overordentlig skidt, hvis meningen er, at eleverne skal lære at skrive tidssvarende og sikker kode.

Prepared Statements
En langt bedre tilgang er at bruge prepared statements (forberedte erklæringer). Her forgår forespørgslen i følgende trin:

1) *Prepare*: Databasens styringsmekanisme opretter en 'skabelon' på baggrund af en SQL-streng, hvor nogle værdier er erstattet af spørgsmålstegn:

```
SELECT `fornavn` FROM `tabel` WHERE `id` = ?
```

Databasen opretter en optimeret plan for forespørgslen og gemmer denne plan - uden at gennemføre forespørgslen.

2) *Bind parameters*: Til den eller de værdier, der var erstattet af spørgsmålstegn i skabelonen, bindes egentlige værdier i form af f.eks. strenge eller tal. I ovenstående eksempel kunne det være tallet **123**.

3) *Execute*: Databasen udfører forespørgslen med de bundne værdier.

Ved forespørgsler, som skal returnere et sæt af resultater/rækker, anvendes desuden følgende to trin:

4) *Bind result*: De fundne data bindes til en eller flere variabler. I eksemplet ovenfor kunne denne variabel f.eks. kaldes $fornavn.

5) *Fetch*: De fundne rækker hentes én ad gangen - lidt, som det kendes fra 'det gamle API'. Når en række er hentet, vil $fornavn (variablen fra resultatbindingen) indeholde værdien af det søgte felt i rækken.

Så længe det pågældende statement (den optimerede plan) ikke lukkes, kan punkt 2, 3, 4 og 5 udføres så mange, man måtte ønske.

Fordelen ved denne fremgangsmåde er, at værdierne ikke sendes til databasen sammen med selve SQL-kommandoen, men sendes som parametre på et senere tidspunkt og ikke kommer i direkte forbindelse med resten af SQL-kommandoen. Databasen tager sig selv af den fornødne escaping af parametrene, hvorved SQL-injection forhindres.

Der er derfor ikke længere brug for at 'hacke' sig frem med mysql_real_escape_string el.lign.

En anden fordel er, at planen kan genbruges, hvilket i høj grad forbedrer performance ved gentagne ens forespørgsler med forskellige værdier. Det skal nævnes, at der ved en enkelt forespørgsel er tale om en marginalt dårligere performance, hvilket dog opvejes af den dramatisk forbedrede sikkerhed.

Kodeeksempler
For at åbne en forbindelse til databasen bruges følgende kode, som i høj grad ligner det, vi allerede kender. Dog defineres databasen med det samme, når forbindelsen etableres:

```
$mysqli = new mysqli("server", "brugernavn", "kodeord", "databasenavn");

/* Tjek, om der opstod en fejl */
if (mysqli_connect_errno()) {
```

```
    echo 'Der opstod en fejl ved forbindelsen: ' . mysqli_connect_error();
    exit();
}
```

Eksempel på en SELECT forespørgsel, hvor id er større en værdi, hentet fra dokumentets URL:

```
/* Opret et prepared statement */
if ($stmt = $mysqli->prepare('SELECT `fornavn`, `efternavn` FROM `tabel` WHERE
`id` > ?')) {

    /* Bind parametre */
    $stmt->bind_param('i', $id);

    /* Sæt værdier på parametrene */
    $id = $_GET['id'];

    /* Eksekver forespørgslen */
    $stmt->execute();

    /* Bind resultatet */
    $stmt->bind_result($fornavn, $efternavn);

    /* Hent rækker og udskriv data */
    while ($stmt->fetch()) {
        echo $fornavn . ' ' . $efternavn . '<br>';
    }

    /* Luk statement */
    $stmt->close();

} else {
    /* Der er opstået en fejl */
    echo 'Der opstod en fejl i erklæringen: ' . $mysqli->error;
}
```

NB: Læg mærke til udtrykket:

```
$stmt->bind_param('i', $id);
```

I et kald til metoden **bind_param** består første argument altid af en streng med ligeså mange tegn, som der er efterfølgende parametre. I dette eksempel kun med ét: '**i**' - men i det følgende eksempel med flere tegn. Disse tegn skal modsvare de pågældende variablers type - og kan antage følgende værdier:

*) **s:** string
*) **i:** integer
*) **d:** double
*) **b:** blob (binære data, sendt som pakker)

Eksempel på INSERT fra form med *method*="*post*":

```
/* Opret et prepared statement */
if ($stmt = $mysqli->prepare('INSERT INTO `tabel` (`fornavn`, `efternavn`,
`stamp`) VALUES (?, ?, ?)')) {

    /* Bind parametre */
    $stmt->bind_param('ssi', $fornavn, $efternavn, $stamp_now);

    /* Sæt værdier på parametrene */
    $fornavn = $_POST['fornavn'];
    $efternavn = $_POST['efternavn'];
    $stamp_now = time();

    /* Eksekver forespørgslen */
    $stmt->execute();

    /* Luk statement */
    $stmt->close();

} else {
    /* Der er opstået en fejl */
    echo 'Der opstod en fejl i erklæringen: ' . $mysqli->error;
}
```

Eksempel på UPDATE fra form med *method*="*post*":

```
/* Opret et prepared statement */
if ($stmt = $mysqli->prepare('UPDATE `tabel` SET `fornavn`=? WHERE `id`=?')) {

    /* Bind parametre */
    $stmt->bind_param('si', $fornavn, $id);

    /* Sæt værdier på parametrene */
    $fornavn = $_POST['fornavn'];
    $id = $_POST['id'];

    /* Eksekver forespørgslen */
    $stmt->execute();

    /* Luk statement */
    $stmt->close();

} else {
    /* Der er opstået en fejl */
    echo 'Der opstod en fejl i erklæringen: ' . $mysqli->error;
}
```

Vigtige overvejelser

I de viste eksempler er brugt $_GET og $_POST variabler uden forudgående validering. Det er kun gjort for overskuelighedens skyld. Man bør altid først validere input 'udefra' - altså data, som ikke er skrevet ind i

koden af udvikleren selv.

Mange tænker i denne forbindelse ikke på, at en $_COOKIE variabel *også* kommer udefra. Den stammer som bekendt fra en tekstfil, der er gemt på brugerens PC og kan derfor meget vel være manipuleret.

Hvis du forventer en variabel skal indeholde et tal, så tjek, om variablen faktisk indeholder et tal - og tjek også, om det er indenfor forventede minimums- og maksimumsværdier. Forventer du en streng, så tjek, om det er en streng, og om den passer med det format, du forventer, osv.

OBS: Hvad udskrivning af fejlmeddelelser angår, så er de her viste eksempler *kun* ment som eksempler, beregnet for læreprocessen. I virkeligheden udgør de i sig selv en sikkerhedsrisiko, hvis de får lov at ligge på en offentlig tilgængelig server. Jeg vil derfor varmt anbefale, at du læser min guide om sikrere håndtering af MySQL-fejl.

Forslag til videre læsning
PHP-manualen om MySQLI
PHP-manualen om MySQLI klassens properties og metoder
PHP-manualen om statement klassens properties og metoder
PHP-manualen om result klassens properties og metoder

Kommentar af michael_stim d. 01. mar 2012 | 1

Dejlig letforståelig gennemgang, har allerede henvist til denne guide, og kommer med garanti at gøre det mange gange til. Er lettere at henvise hertil, end at skulle til at forklare hver gang.

Kommentar af olebole d. 01. mar 2012 | 2

Tak Michael - og du rammer min egen begrundelse for at skrive den lige på sømmet! *o)

Kommentar af jakobdo d. 07. mar 2012 | 3

Så forstod selv jeg MySQLi. :o)

Kommentar af Stefan1 d. 17. mar 2012 | 4

Er det også muligt at bruge update i Prepared Statements?

Kommentar af olebole d. 19. mar 2012 | 5

@**Stefan1**: Jeg har tilføjet et UPDATE eksempel =)

Kommentar af Stefan1 d. 19. mar 2012 | 6

@Ole Mange tak Ole.

Kommentar af h_thunbo d. 20. mar 2012 | 7

Den bliver lige tilføjet til favoritter indtil det sidder på rygraden :-)

Kommentar af tobrukDk d. 29. mar 2012 | 8

Hej

jeg forstår det ikke , er der nogle som kan forklare mig det, jeg har prøve frem og tilbage og jeg kan bare

ikke få det til at virker overhovedet :O

Kommentar af michael_stim d. 30. mar 2012 | 9

#8
Øhhh, HVAD skal vi forklare? Den forklaring du har læst? Det er svært at forklare tydeligere end hvad der er skrevet. Opret et spørgsmål og forklar hvor i din kode, det er det går galt.

Kommentar af tobrukDk d. 23. apr 2012 | 10

#9 efter som har snakke med olebole privat via PM så har han forklare mig det på en måde så jeg forstår det på :) Lækkert lækkert og det er pisse godt lavet ole! +1!!

Kommentar af tobrukDk d. 21. maj 2012 | 11

Jeg synes personligt at Mysqli det er langt bedre endnu det "gammel"

Kommentar af oomalkeoo d. 15. jun 2012 | 12

Endnu en perfekt guide fra dig, mange tak!

Kommentar af tobrukDk d. 20. jun 2012 | 13

#olebole. Du mangler også at lave en delete function/hvad man kalder det..

Kommentar af Jacobmoller d. 20. jul 2012 | 14

Tak for denne letlæselige og forståelige guide.

Jeg har lige et spørgsmål - jeg vil gerne bruge den i forbindelse med at oprette en bruger, og der bruger jeg MD5 til password.

Hvad hedder det i bind_param? Er det bare en integer?
Altså
$stmt->bind_param('???', $password)

Kommentar af DeeDawg d. 20. jul 2012 | 15

@**Jacobmoller**: Nej, det vil gå som en string. At du ikke bør benytte MD5, er en anden side af sagen. :)

Kommentar af olebole d. 20. jul 2012 | 16

```
$stmt->bind_param('s', $password);
```

Som DeeDawg (næste) skriver, bør du nok bruge sha1 eller hash[/b] (f.eks. med sha256 - som dog fylder væsentligt mere). Desuden bør du altid 'salte' hash'en. **md5** er for udbredt i de såkaldte [url=http://en.wikipedia.org/wiki/Rainbow_table]rainbow tables.

Kommentar af kjeldsted d. 31. jul 2012 | 17

Lækker skrevet guide.
Og bestemt noget jeg da skal til at begynde på (ikke mindst efter "advarslen" fra PHP mod at benytte

mysql APIen).

Kommentar af tobrukDk d. 12. aug 2012 | 18

Nu komme jeg til at tænke på hvad er overhovedet bedste MD5 eller sha1? og hvorfor?

Kommentar af arne_v d. 14. aug 2012 | 19

MD5 har vaeret foraeldet i flere aar.

SHA1 er foraeldet idag.

SHA2 f.eks.SHA-256 er hvad du skal bruge.

Kommentar af havemaskiner d. 05. sep 2012 | 20

De guider her! De er godt nok perfekte.

Kommentar af mccookie d. 18. sep 2012 | 21

Har jo brugt det gamle API og en masse escapes mv....

Men det her er da egentligt rimeligt at gå til :-)

God guide - Let forklarligt og lige ud ad landevejen....

Kommentar af Stefan1 d. 15. okt 2012 | 22

Hvordan tilføjer man LIKE statements med wildcards hvis man vil sammenligne med to kolonner fra tabellen?

Kommentar af olebole d. 17. okt 2012 | 23

Hvis det er en variabel, du vil bruge, og du bruger prepared statements:

```
$stmt = $mysqli->prepare('SELECT `id` FROM `foobar` WHERE `foo` LIKE ?');
$parameter = '%' . $someVar . '%';
$stmt->bind_param('s', $parameter);
```

Hvis du vil sammenligne to felter (foo og bar):

```
$res = $db->query("SELECT `id` FROM `foobar` WHERE `foo` LIKE CONCAT('%', `bar`, '%')");
```

Kommentar af olebole d. 17. okt 2012 | 24

PS: I det andet eksempel, skal du passe på. Tegnet '%' i indholdet af feltet 'bar' vil skabe uventede resultater

Kommentar af Stefan1 d. 19. okt 2012 | 25

Tak ole :)

Kommentar af skabsforhandler d. 02. feb 2013 | 26

Super god guide. Nem at forstå, selv for en som ikke er så stærk til kodning (Læs, Mig) :-) Tak for hjælpen

Kommentar af riefart d. 10. mar 2013 | 27

Syntes, at jeg lige havde fået en lille smule hold på det gamle API, og var lidt uvillig til at gå i gang med PS. Men for filen Ole, som du forklarer det, er det både logisk og enkelt.
Tak for en god guide.

Kommentar af Peter89 d. 03. apr 2013 | 28

Jeg har besluttet mig for at gå fra det gamle mysql_ API til mysqli, og jeg synes denne guide har hjulpet mig godt i gang - så tak for det! :-)

Jeg oplever dog denne fejl i min implementation, hvilket jeg undrer mig lidt over.

Fatal error: Allowed memory size of 134217728 bytes exhausted (tried to allocate 4294967296 bytes)

Er der nogen der har et bud på hvad det kan skyldes?

Kommentar af nemlig d. 08. apr 2013 | 29

Super guide, som giver en god og grundlæggende indsigt i principperne. Jeg er hermed kommet godt i gang med at omkode mine scripts.

Kommentar af olebole d. 23. apr 2013 | 30

Tak for rosen til alle =)

@Peter89: Den fejl må vist stamme fra noget andet

Kommentar af kennethhede d. 01. aug 2014 | 31

Super god guide! Har selv haft en del problemer med MySQL på Sportsway, men har fået det fikset nu :-)