Perl und Datenbanken

Perl muss in vielen Fällen mit Datenbanken arbeiten. Datenbanken werden für dynamische Webseiten benötigt oder in den meisten Fällen, in denen riesige Datenmengen verarbeitet werden müssen.

Perl bietet - wie so häufig - verschiedene Wege, um mit einer Datenbank arbeiten zu können. In diesem Artikel wird das Beispielhaft mit MySQL gemacht, wobei es für (fast) alle Datenbanken einen entsprechenden Treiber gibt. Das Zauberwort für die Arbeit mit Datenbanken heißt `DBI`.

`DBI` bedeutet *DataBase Independent Interface* und ist genau das - eine Schnittstelle unabhängig vom dahinterliegenden Datenbanksystem (*DBMS*). Man kann sagen, dass `DBI` als Proxy dient. Es leitet die Anfragen beziehungsweise die Funktionsaufrufe direkt an den Datenbanktreiber weiter wie in Abbildung 1 zu sehen ist.
In `DBI` ist auch festgelegt, welche Funktionen ein solcher Treiber implementieren muss.

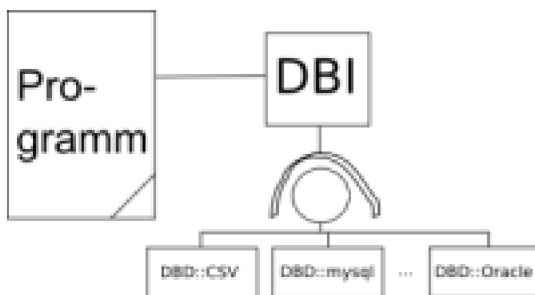

Abbildung 1: DBI als "Proxy" für Datenbanktreiber

Datenbanktreiber

Für die gängigen Datenbanksysteme gibt es einen Treiber, der mit `DBI` verwendet werden kann. Diese Treiber können auf `CPAN` gefunden werden. Einige ausgewählte Treiber sind *DBD::mysql*, *DBD::Oracle*, *DBD::ODBC* und *DBD::CSV*. Selbst auf CSV-Dateien und Excel lässt sich über `DBI` zugreifen.
Die Treiber sind für den DBMS-spezifischen Teil zuständig.

Arbeit mit der Datenbank

Nach den theoretischen Grundlagen geht es jetzt in die praktische Arbeit mit Datenbanken. In den Beispielen wird mit MySQL und dem entsprechenden Treiber DBD::mysql gearbeitet.

Verbindung herstellen

In Listing 1 wird gezeigt, wie die Verbindung zu einer MySQL-Datenbank aussehen würde.

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use DBI;
 6
 7 my $db    = 'NameDerDatenbank';
 8 my $user  = 'Datenbank_User';
 9 my $pass  = 'Passwort';
10 my $host  = 'localhost';
11 my $dsn   = "DBI:mysql:$db:$host";
11
12 my $dbh   = DBI->connect($dsn,
13                $user,$host)
14                or die $DBI::errstr;
15
16 print "Verbindung hergestellt\n";
```

Listing 1: Verbindung zur Datenbank

Die Zeile 12 ist auch die einzige, die bei Verwendung einer anderen Datenbank angepasst werden muss. So kann zum Beispiel die Verbindung zu einer SQLite-Datenbank hergestellt werden - wenn die Zeile 12 durch folgende Zeile ersetzt wird

```
my $dbh = DBI->connect("DBI:SQLite...")
            or die $DBI::errstr;
```

Wenn die Verbindung zur Datenbank hergestellt ist, können verschiedene SQL[2]-Abfragen an die Datenbank gestellt werden. Wie dies aussieht, wird in den folgenden Absätzen erläutert.

Tabellen erzeugen

In diesem Beispiel sollen zwei Tabellen angelegt werden. In der Grafik 2 ist zu sehen, wie die Tabellen aussehen. Die dazugehörigen *CREATE*-Statements wurden mit Hilfe des Moduls `FabForce::DBDesigner4` ermittelt. Um die Tabellen anlegen zu können, wird zuerst wieder eine Verbindung zur Datenbank aufgebaut. Danach wird die der SQL-Befehl "vorbereitet" und ausgeführt. Listing 2 zeigt, wie das Skript dann aussieht.

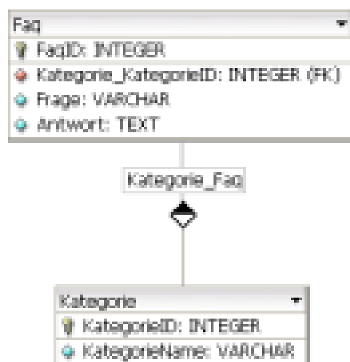

Abbildung 2: Beispieltabellen

Durch die do-Methode von `DBI` wird der SQL-Befehl ausgeführt und die Tabellen werden erstellt.

```
 1 #!/usr/bin/perl
 2
 3 use strict;
 4 use warnings;
 5 use DBI;
 6
 7 my $db_name = 'NameDerDatenbank';
 8 my $db_user = 'Datenbank_User';
 9 my $db_pass = 'Datenbank_Passwort';
10 my $db_host = 'localhost';
11
12 my $dbh     = DBI->connect("DBI:mysql:$db_name:$db_host",
13                            $db_user,$db_host)
14                            or die $DBI::errstr;
15
16 my $faq     = qq~CREATE TABLE Faq(
17          FaqID        INT           NOT NULL PRIMARY KEY,
18          Frage        VARCHAR(255)  NOT NULL,
19          Antwort      TEXT          NOT NULL,
20          KategorieID  INT           NOT NULL,)~;
21
22 my $cat     = qq~CREATE TABLE Kategorie(
23          KategorieID   INT           NOT NULL PRIMARY KEY,
24          KategorieName VARCHAR(100)  NOT NULL,)~;
25
26 for($faq,$cat){
27     my $sth     = $dbh->do($_) or die $dbh->errstr();
28 }
```

Listing 2: Anlegen von Tabellen mit DBI

Tabelle auslesen

In den meisten Fällen sollen Daten aus der Datenbank ausgelesen werden. Für diese Fälle werden *SELECT*-Befehle benötigt. Vom Prinzip her wird jetzt das gleiche gemacht wie in den anderen Beispielskripten auch. Jetzt werden die Ergebnisse allerdings abgefragt und ausgegeben.

Dazu gibt es verschiedene `fetch*`-Methoden. Im Listing 3 werden die `fetchrow_array`- und die `fetchrow_hashref`-Methode gezeigt.

andere Befehle

Genauso einfach wie die bisher gezeigten Beispiele, können Datensätze eingetragen oder aktualisiert werden, Tabellen geändert oder gelöscht werden. Für Funktionen, die Datenbank-spezifisch sind - wie zum Beispiel Stored Procedures - kann die DBI-Methode C<func> verwendet werden.

Ein einfaches Beispiel für C<INSERT>s sieht folgendermaßen aus:

```
1 my $insert = q~INSERT INTO
2             Kategorie VALUES(?,?)~;
3 $dbh->do($insert,undef,1,
4             'Neue Kategorie');
```

Sicherheit

Gerade bei Webanwendungen spielt die Sicherheit eine sehr große Rolle. Hacker versuchen mit sogenannten SQL-Injections Informationen über die Datenbank und die Inhalte zu bekommen. `DBI` bietet aber ein Feature, mit dem die Sicherheit der

```
 1 # fetchrow_array Beispiel
 2
 3 my $stmt = "SELECT * FROM persons";
 4 my $sth  = $dbh->prepare($stmt);
 5 $sth->execute();
 6
 7 while( my @row = $sth->fetchrow_array() ){
 8     print join(";",@row);
 9 }
10
11 # fetchrow_hashref Beispiel
12
13 my $stmt2 = "SELECT * FROM Faq"
14 my $sth2  = $dbh->prepare($stmt2);
15 $sth2->execute();
16
17 while( my $hashref = $sth2->fetchrow_hashref() ){
18     print "Datensatz:\n";
19     while( my ($key,$value) = each %$hashref ){
20         print "Spalte $key: $value\n";
21     }
22 }
```

Listing 3: SELECT-Statements

Perlskripte stark erhöht werden kann: die ?-Notation. Die gleiche Wirkung kann auch mit der DBI-Methode quote erreicht werden, aber die ?-Schreibweise ist kürzer und kann für Optimierungen verwendet werden.
Das folgende Codefragment zeigt die ?-Schreibweise für ein *SELECT*-Statement.

```
1 my $vorn  = "Tim";
2 my $nachn = "O'Reilly"
3 my $stmt  = q~SELECT * FROM
4     persons WHERE firstname = ?
5     AND name = ?~;
6 my $sth = $dbh->prepare($stmt);
7 $sth->execute($vorn,$nachn);
```

Durch das quote oder die ?-Notation werden Sonderzeichen gequoted damit es bei der SQL-Abfrage keine Probleme gibt.
Zum Beispiel der Nachname "O'Reilly" macht Probleme wenn man String-Werte im SQL einfach mit ' umgibt wie in

```
my $stmt = q~SELECT * FROM persons
     WHERE nachname = '$nachname'~;
```

Wenn dieses Statement ausgeführt wird, tritt ein Fehler auf, weil das ' in "O'Reilly" nicht geschützt ist. Um diesen Fehler zu vermeiden gibt das quote und die ?-Schreibweise. Diese zwei Möglichkeiten sind in Listing 4 gezeigt.

```
 1 # mit quote
 2 my $nachname = "O'Reilly";
 3 $nachname    = $dbh->quote($nachname);
 4 my $stmt     = qq~SELECT * FROM persons
 5       WHERE nachname = '$nachname'~;
 6 my $sth      = $dbh->prepare($stmt);
 7 $sth->execute();
 8
 9 # mit ?
10 my $lastname = "O'Reilly";
11 my $stmt2    = q~SELECT * FROM persons
12        WHERE nachname = ?~;
13 my $sth      = $dbh->prepare($stmt2);
14 $sth->execute($lastname);
```

Listing 4: ?-Notation

Zurück zu den SQL-Injections. Als Beispiel, wo solche Angriffe auftauchen können ist der Login zu einem Admin-Bereich einer Webseite. Viele solcher Login-Skripte im Internet verwenden einen SQL-Befehl wie

```
SELECT count(*) FROM
myusers WHERE id='$name'
AND password='$password'
```

und wenn count() mehr als 1 Element zurückliefert, ist der Nutzer eingeloggt.
Ein Angreifer könnte als ID

```
' OR '1' = '1
```

und das gleiche als Passwort eingeben. Zusammengesetzt sieht dann die SQL-Abfrage so aus:

```
SELECT count(*) FROM myusers WHERE
id='' OR '1'='1' AND password='' OR
'1'='1'
```

Und schon ist der Angreifer im Admin-Bereich. Mit der ?-Notation wäre das nicht passiert, weil dann die ' in den Werten, die vom Angreifer eingegeben wurden, geschützt sind.

Optimierung

Die ?-Schreibweise erhöht nicht nur die Sicherheit, sondern kann auch zur Perfomance-Steigerung verwendet werden.
Viele Skripte sehen wahrscheinlich so aus wie der folgende Code.

```
1 my @daten = qw(dies ist ein Test);
2 for my $value(@daten){
3   my $sth = $dbh->prepare("INSERT
      INTO tabelle VALUES('$value')");
4     $sth->execute();
5 }
```

Das kostet Zeit, weil bei jedem Schleifendurchlauf die SQL-Abfrage "vorbereitet" wird und dann erst ausgeführt wird.
Mit der ?-Schreibweise kann man das beschleunigen, da die SQL-Abfrage immer gleich ist (bis auf den einzufügenden Wert). Diese Beschleunigung funktioniert nicht nur mit *INSERT*s, sondern mit allen Abfragen. Optimiert sieht der Codeausschnitt folgendermaßen aus.

```
1 my @daten = qw(dies ist ein Test);
2 my $stmt  = "INSERT INTO tabelle
                 VALUES(?)"
2 my $sth   = $dbh->prepare($stmt);
3 for my $value(@daten){
4    $sth->execute($value);
5 }
```

Die Fehlerabfrage ist aus Gründen der Übersichtlichkeit weggelassen.

Fallen

Hier werden zwei "Fallen" dargestellt, in die man leicht tappen kann und bei denen die Fehlerfindung nicht ganz einfach ist. Bei diesen Fallen, bleibt der Code nahezu unverändert nur den String bei der Herstellung der Verbindung muss angepasst werden.

MS-Access und .mdw-Dateien

Viele Access-Datenbanken sind durch eine Passwort-Datei geschützt. Diese Dateien enden mit .mdw. Wenn Access diese Datei nicht genannt bekommt, kann man keine Verbindung zu der Datenbank herstellen.
Dazu wird eine Workgroup-Datei anglegt - die .mdw-Datei. Darin werden die Gruppen und die Berechtigungen gespeichert. Mit einer "normalen" Verbindung wird das ganze jetzt fehlschlagen und man wird gebeten, den Administrator der Datenbank um die Rechte zu bitten. Eine kleine Änderung im DSN[1]-String bewirkt aber, dass der Verbindungsaufbau funktioniert.
Im DSN-String muss noch der Key "SystemDB" auftauchen mit dem Pfad zur .mdw-Datei:

```
1 my $dsn  ='driver=Microsoft Access-
  Driver (*.mdb); dbq=c:\database.mdb;
  SystemDB=c:\sicherheit.mdw';
2 my $dbh = DBI->connect(
        "DBI:ODBC:$dsn", $user,$pass)
        or die $DBI::errstr;
```

Oracle ohne Umgebungsvariablen

In manchen Fällen sind die Umgebungsvariablen für Oracle nicht gesetzt - dann findet der Datenbanktreiber die Datenbank nicht und eine Verbindung kann nicht hergestellt werden. Auch in diesem Fall muss einfach der DSN-String angepasst werden:

```
1 my $dsn ='host=myhost.com;sid=ORCL';
2 my $dbh = DBI->connect(
       "DBI:Oracle:$dsn",$user,$pass)
       or die $DBI::errstr;
```

In diesem Fall erzeugt `DBD::Oracle` den vollen Descriptor-String und benötigt nicht tsnames.ora.

Lösung **ohne** DBI

C<DBI> ist die praktischste Lösung, um mit Datenbanken zu arbeiten. Es gibt allerdings auch Module, die neben `DBI` existieren. Eine häufig genutzte Variante ist das Arbeiten mit `Win32::ODBC`. Dennoch ist `DBI` in mindestens 95% aller Fälle die bessere Wahl.

Referenzen

http://perl.renee-baecker.de/perl_datenbanken.pdf
http://search.cpan.org/dist/DBI/
http://www.perl.com/pub/1999/10/DBI.html
http://perloo.de/DBI/
http://www.northbound-train.com/perl/article/Article.html
http://aktuell.de.selfhtml.org/artikel/cgiperl/odbc/
http://de.wikipedia.org/wiki/SQL_Injection

[1] Data Source Name
[2] Structured Query Language