© Kiskapu Kft. Minden jog fenntartva

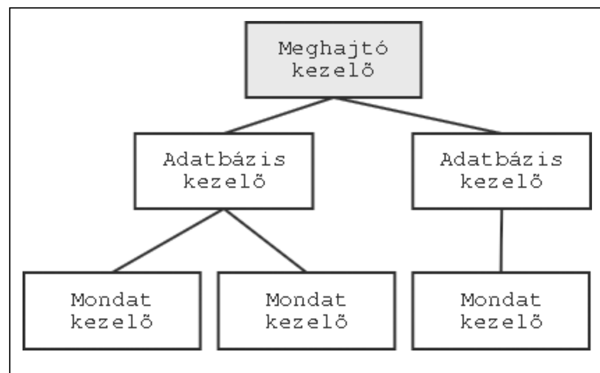
A Perl és az adatbázisok (3. rész)

Hajtsuk uralmunk alá a világ összes adatbáziskezelőjét SQL parancsokkal! A Perl DBI modulja segít nekünk ebben.

Az előző részekben megismerkedtünk az adattárolás alapjait jelentő szöveges, és a ***Berkeley*** alrendszerre támaszkodó adatbázisokkal. Ezek programozása elég egyszerű és gyors, ám nagyobb rendszerek esetén komoly nehézségek adódhatnak ha ezeket használjuk. A sokféle kódlap miatt az adatállományok nehezen hordozhatóak, a mezők típusai csak körülményesen adhatók meg, a hálózati elérésről pedig ne is beszéljünk.
Ezek a gondok abból adódnak, hogy az első részben hangoztatott adatfüggetlenítési elvet eddig egyáltalán nem vettük figyelembe. A fent említett feladatokat ugyanis egy elkülönített adatkezelő rétegnek kellene ellátnia. Az eddig bemutatásra került módszerekkel ezt nem lehetett megvalósítani, most azonban lépünk egy nagyot, és megnézzük, hogyan lehet egyszerre akár több relációs adatbáziskezelő rendszert (***RDBMS***) is használni.

Közös nyelv – közös felület

Feltételezem, hogy az Olvasó találkozott már olyan táblákból, és az azok közötti kapcsolatokból építkező adatbáziskezelővel, amelynek a kezelőnyelve az ***SQL*** volt. Ebben a hónapban azt vizsgálom meg, hogyan használhatunk egy ilyen rendszert a ***Perl*** segítségével. Persze nem tudhatom, hogy az Olvasó ***MySQL***, vagy ***PostgreSQL*** párti. Ha azonban egységesített felületen keresztül érjük el kedvenc adatbázisunkat, és a háttérben dolgozó meghajtó-'programra bízzuk az egyedi sajátságok kezelését, akkor ez nem is lényeges.
A közös felületet a ***DBI*** jelenti. Ez egy ***Perl*** modul formájában megjelenő programozói felület, amely mögött számos elterjedt adatbáziskezelő rendszerhez találunk meghajtóprogramot. Röviden szólva a ***DBI*** használatával egy újabb réteg kerül az adatbáziskezelő és saját programunk közé, ehetővé téve ezáltal hogy úgy álljunk át például ***MySQL***-ről ***Oracle***-re, hogy ehhez a program akárcsak egyetlen sorát át kellene írnunk. Azt hiszem ezt hívják programozói álomnak.
Minden támogatott rendszert egy-egy ***DBD*** modul képvisel, ezek jelentik a ***DBI*** mögötti meghajtókat. A rövidítések a belső logikai kapcsolatrendszerre utalnak: ***DBD*** = „***DataBase Dependent***” (adatbázisfüggő) és ***DBI*** = „***DataBase Independent***” (adatbázistól független). Jelenleg az alábbi rendszerekhez van megbízható támogatás: ***Oracle***, ***Informix***, ***mSQL***, ***MySQL***, ***Ingres***, ***Sybase***, ***DB2***, ***Empress***,



SearchServer, ***PostgreSQL***, és ***XBase***. A régi idők szerelmeseinek pedig van egy ***CSV*** modul is a közönséges szöveges állományokhoz.
Az egyes adatbáziskezelők meghajtóit tartalmazó modulok `DBD::rdbms` néven érhetők el, ahol az `rdbms` a szóban forgó rendszer neve. Így az ***Oracle*** meghajtót a `DBD::Oracle` tartalmazza. A ***DBI*** elég okos ahhoz, hogy a csatlakozáskor megadott adatforrás alapján önműködően betöltse a megfelelő meghajtót, így erről nekünk már nem kell gondoskodnunk. Nem csak hogy nem kell, valójában nem is szabad, így az egyetlen modul, amit nekünk, a programozónak be kell töltenünk, maga a ***DBI***.

A DBI felépítése

A ***DBI*** teljesen objektumközpontú. Némi túlzással fogalmazhatunk úgy, hogy a ***DBI*** felépítése az egységbezárás mintapéldája. Ennek megvan az a szépsége, hogy lehet akár párhuzamosan két kapcsolatunk két különböző adatbáziskezelővel, az ezeken elvégezhető műveletek biztosan nem fognak keveredni. Kevésbé pongyolán fogalmazva, tiszta és átlátható programot kapunk az elemfüggvények használatával.
Három fő objektumtípus áll rendelkezésünkre a kedvenc adatbáziskezelőnkkel történő kapcsolattartásra. Ezeket ***kezelőknek*** (***handle***) hívják, és egy jól meghatározott hierarchiát alkotnak. Az első a ***meghajtó kezelő*** (***driver handle***), amely egy belső használatra fenntartott objektum, és egy betöltött meghajtót jelképez. Nekünk ezzel nem kell foglalkoznunk, a `connect()` elemfüggvényt is az ősosztályon keresztül fogjuk meghívni.

A második az ***adatbázis kezelő*** (***database handle***), amely egy pillanatnyi munkamenetet jelképez egy adott adatbázissal. Ez a hozzá tartozó meghajtó kezelő gyermeke és mi rendszerint a `DBI -> connect()` függvénytől kapjuk vissza. Ez utóbbi a paraméterként átadott adatforrásból fejti meg, hogy mely meghajtóból kell származtatni az adatbázis kezelőt. A ***DBI*** leírása és a példa programok alapján `$dbh` néven szokás hivatkozni rá.
Végül, de nem utolsó sorban a ***mondatkezelő*** (***statement handle***) egy ***SQL*** mondatot zár egységbe. A mondat kezelők az adott adatbáziskezelő gyermekei. Mint ilyenek, egy mondat kezelő adatai védettek a többi mondat kezelő módosításaitól. Rendszerint a szülő `prepare()` elemfüggvényétől kapjuk vissza ezt az objektumot. A fentebb bevezetett elnevezési módot követve, `$sth` névvel fogunk hivatkozni rá.

Az adatforrások

Amikor egy adatbázishoz csatlakozunk a ***DBI*** segítségével, meg kell határoznunk egy úgynevezett adatforrást. Ez egy kettőspontokkal elválasztott lista. Első tagja kötött, és mindig a `DBI` betűk alkotják, akárcsak a honlapok elérésénél használt `http`. Második tagja a használni kívánt meghajtó neve, pl. `Xbase`. A további paraméterek mind átadásra kerülnek a meghajtó kezelő saját `connect()` függvényének. Ezek már meghajtónként eltérhetnek, itt szokás megadni pl. azt TCP/IP kaput, amin az adatbáziskezelő figyel.
Első körben próbáljuk meg feltérképezni saját rendszerünk adta lehetőségeket. A ***DBI*** osztály `available_drivers()` elemfüggvénye egy tömböt ad vissza, amely a használatra kész meghajtókat tartalmazza. Ezek bármelyikét átadva a `data_sources()` elemfüggvénynek, megkapjuk az adott meghajtón keresztül igénybe vehető adatforrások listáját, szintén egy tömbként. Ezek segítségével megjeleníthetjük az összes meghajtó összes adatforrását az adott rendszeren:

```
#!/usr/bin/perl -w

use strict;
use DBI;

my @meghajtok = DBI -> available_drivers ();
die "Nem találtam meghajtókat!\n" unless
     ➥@meghajtok;

foreach my $meghajto ( @meghajtok ) {

    print "Meghajtó: " . $meghajto . "\n";
    my @adatforrasok = DBI -> data_sources
       ➥( $meghajto );

    foreach my $adatforras ( @adatforrasok ) {
       print "\tAdatforrás: " . $adatforras .
             ➥"\n";
    }

    print "\n";

}
```

Úgy gondolom, a forrás önmagáért beszél. Én a következő kimenetet kaptam:

```
Meghajtó: ExampleP
       Adatforrás: dbi:ExampleP:dir=.

Meghajtó: Proxy

Meghajtó: mysql
       Adatforrás: DBI:mysql:mysql
       Adatforrás: DBI:mysql:mythconverg
       Adatforrás: DBI:mysql:proba_adatbazis
       Adatforrás: DBI:mysql:test
```

Elég szegényes telepítés, mindössze három meghajtóm van! Ezek sorát a megfelelő `DBD::meghajto` modul kézi telepítésével, vagy ***Debian*** alatt ***libdbd-meghajto-perl*** csomag beszerzésével lehet bővíteni. Azt azonban érdemes megfigyelni, hogy a ***MySQL*** adatforrások között az összes ***MySQL*** adatbázisomat látom. Ez azt is jelenti, hogy az adatbázisok neveire mindennemű hitelesítés nélkül, egyszerűen szert lehet tenni.

Lányok lekérdezése

Következő lépésként lássunk egy kis szkriptet, amely egy, a már a korábbi részekből megszokott lekérdezést valósít meg egy ***MySQL*** adatbázison. Látni fogod, hogy az adatbázishoz szorosan kötődő kód mindössze egy sor, így a program könnyen átültethető egy másik környezetbe.

```
#!/usr/bin/perl -w

use strict;
use DBI;

die "Használat: " . $0 . " <~lány neve>\n" unless
➥@ARGV;

my ($dbh, $sth, $minta, @sor);

$dbh = DBI -> connect (
    "DBI:mysql:proba_adatbazis",
    "testuser",
    "jelszo",
    {
       PrintError => 0,
       RaiseError => 1
    }
);

$minta = $dbh -> quote ($ARGV [0]);
$sth = $dbh -> prepare ("SELECT * FROM lanyok
➥WHERE nev LIKE " . $minta);

$sth -> execute ();

while (@sor = $sth -> fetchrow_array ()) {
    my ($nev, $telszam, $szereti) = @sor;
    print " Adatlap: " . $nev . "\n";
```

© Kiskapu Kft. Minden jog fenntartva

© Kiskapu Kft. Minden jog fenntartva

```
    print "==========" . "=" x length ( $nev ) .
        ➥"\n";
    print "  Telefonszám               : " .
        ➥$telszam . "\n";
    print "  Ezzel veheted le a lábáról : " .
        ➥$szereti . "\n\n";
}

$dbh -> disconnect ();
```

Nézzük sorról-sorra! Az első sorban feltüntetjük a parancsértelmező elérési útját, és jelezzük a -w kapcsolóval, hogy a figyelmeztető üzeneteket is szeretnénk látni. Bekapcsoljuk a szigorú ellenőrzést, majd használatba vesszük a ***DBI*** modult. A negyedik sorban egy, a program helyes használatára vonatkozó üzenettel kilépünk, amennyiben nem volt átadott paraméter. Remélem, az eddig látottak nem okoztak meglepetést.
Az ötödik sorban jelezzük a használni kívánt változókat. A `$dbh` az adatbázis kezelő, a `$sth` pedig a mondat kezelő lesz. Láthatólag nem lesz szükségünk meghajtó kezelő objektumra, hiszen azt kizárólag belső használatra szánták.
A $minta a parancssorban átadott mintát fogja tartalmazni, de ezen még finomítunk. Továbbá amikor egy ciklusban végigmegyünk az ***SQL*** lekérdezés eredménylistáján, egy `@sor` tömbben lesz a pillanatnyi rekord.
Ezután meghívjuk a `DBI` osztály `connect()` függvényét, és a visszaadott adatbázis kezelő objektumot a `$dbh` változóban tároljuk. Az elemfüggvény első paramétere az adatforrás, mely kettőspontokkal elválasztva tartalmazza a kötelező `DBI` szót, a meghajtó nevét, és a használni kívánt adatbázist. Ez utóbbit már a meghajtó saját `connect()` függvénye kapja meg. Az adatforrást követő két paraméter tartalmazza a felhasználónevet és jelszót.
A negyedik paraméter egy asszociatív tömb. Az adatbáziskezelővel történő kapcsolattartás egészére vonatkozó attribútumok adhatók meg itt, de az alapértelmezések elfogadása esetén el is hagyható a paraméter. Ha a `PrintError` 1, bármilyen hiba fellépése esetén a ***Perl*** `warn()` függvényén keresztül a felhasználó értesül a hibáról. Alapértelmezésben 1. Ha a `RaiseError` 1, bármilyen hiba fellépése esetén a ***Perl*** `die()` függvényén keresztül hibaüzenet után azonnal kilép. Alapértelmezésben 0.
A program az adatbáziskezelőre bízza a mintaillesztést. Ennek előnye, hogy a kiszolgálónak kell szűkítenie a kiválasztás eredményét, viszont hátránya, hogy sajnos nem lehet szabványos kifejezéseket használni az ***SQL***-ben. Mindenesetre bármilyen szöveget adunk is át az adatbáziskezelőnek, érdemes levédeni. A `quote()` elemfüggvény nem csak két idézőjel közé teszi a teljes szöveget, de a közbeeső különleges karakterekre is figyel.
A következő sorban meghívjuk az adatbázis kezelő `prepare()` elemfüggvényét. Ez előkészít egy ***SQL*** mondatot a végrehajtásra. Az előkészítés azt jelenti, hogy létrehoz egy mondat kezelő objektumot. Utóbbinak az `execute()` elemfüggvényét kell meghívni a lekérdezés végrehajtásához. Ez azért hasznos, mert a mondat összeállítása és a lekérdezés nem feltétlenül kell, hogy egy helyen legyen. Továbbá, egy lekérdezés többszöri végrehajtása adhat különböző eredményeket.

A fő ciklusban rekordonként dolgozzuk fel a kapott eredményeket. Ehhez a mondat kezelő `fetchrow_array()` elemfüggvényét használjuk. Ennek segítségével minden iterációban egy újabb rekord mezőit tartalmazza a `@sor` tömb. A ciklus magja az első rész óta nem sokat változott. Egy szép adatlapot ad, ahol a név a megfelelő hosszúságban alá van húzva, ezt követi a másik két mező. A program legvégén megszakítjuk a kapcsolatot az adatbáziskezelővel.

A hárem bővítése

A most következő gyöngyszem a tárolás műveletet valósítja meg. Némileg módosult a hibakezelés rész, továbbá van egy trükk az ***SQL*** utasítás előkészítésében. Mindezek a fortélyok még nagy segítségedre lehetnek, ha adatbázis-programozásra adnád a fejed. Lássuk!

```
#!/usr/bin/perl -w

use strict;
use DBI;

die "Használat: " . $0 . " <adatbazis_nev>\n"
➥unless @ARGV;

my ($dbh, $sth);

$dbh = DBI -> connect (
    "DBI:mysql:" . $ARGV [0],
    "testuser",
    "jelszo",
    { PrintError => 0, RaiseError => 0 } ) or
     ➥die "Nem tudok csatlakozni az adatbázishoz:
     ➥" . $DBI::errstr;

eval {
    $dbh -> {RaiseError} = 1;
    $sth = $dbh -> prepare ("INSERT INTO lanyok
        ➥VALUES (?, ?, ?)");

    print "Neve                       : ";
    my $nev = <STDIN>; chop $nev;
    print "Telefonszám                : ";
    my $telszam = <STDIN>; chop $telszam;
    print "Ezzel veheted le a lábáról : ";
    my $szereti = <STDIN>; chop $szereti;

    $sth -> execute ($nev, $telszam,
    ➥$szereti);
};
warn "Hiba történt a művelet végrehajtása során: "
➥. $@ if $@;
s
$dbh -> disconnect ();
```

Az első újdonság, hogy parancssorban át lehet adni a használni kívánt adatbázis nevét. Ezzel még nem oldjuk meg a világegyenletet, de megkönnyítheti a szkriptet fekete dobozként kezelő felhasználóink életét. Ha még ennél is figyelmesebbek szeretnénk lenni a program használóival,

az első példában bemutatott módszerrel kikereshetjük az elérhető adatbázisok neveit, és felajánlhatjuk őket választási lehetőségként.

Az ötödik sorban ez esetben már csak két változót határoztunk meg előre. Ez valójában ízlés dolga. Egy *C*-hez szokott programozó jobban szereti a program legelején látni a használni kívánt szimbólumokat, egy *C++*-on felnőtt billentyűzet-nyűvő inkább a használat helyére tenné őket. Ebben a példában a további három változó meghatározása közvetlenül a használatba vétel helyén történik. Döntsük el, melyik tetszik jobban, viszont maradj következetes.

A `connect()` függvény kevéssé változott. Természetesen az adatforráshoz hozzá kell fűzni a parancssori paramétert. Viszont figyeld meg, hogy a `RaiseError` értéke most `0`, vagyis nem lép ki önműködően a program hibaüzenettel, ha a művelet nem volt sikeres. Helyette egy kézi hibaellenőrzést végzünk a már megszokott `fv() or die;` szerkezettel. A saját hibaüzenetben felhasználtuk az ősosztály `errstr` elemváltozóját a hiba típusának leírásához.

Ezután egy `eval {}` blokk következik, melynek első utasításában visszaállítjuk a `RaiseError` értékét 1-re. Így amennyiben a blokkon belül felmerül egy hiba, az önműködően meghívott `die()` miatt nem fejezi be a futását a program, hanem csak a blokkot hagyja el. A különleges `$@` változó tartalmazni fogja a hiba leírását, így azt egy önálló hibaüzenet megjelenítéséhez nyugodtan felhasználhatjuk.

Ez a megoldás kis jóindulattal a Java-s `try {} - catch {}` kistestvérének nevezhető.

Az ezt követő előkészítés tartalmaz három kérdőjelet. Ezekre a kérdőjelekre úgy lehet gondolni, mint az *SQL* mondat belső változóira. Előkészítjük ugyan az utasítást, de bizonyos részeket üresen hagyunk. Ezeket a lyukakat később a végrehajtás során betömhetjük. Itt ennek nincs sok jelentősége, hiszen a beolvasás után is előkészíthettük volna az utasítást. Egy ciklusba ágyazott `execute()` esetén viszont nagyon hasznos lehet ez a módszer.

Az adatok bekérése ugyanúgy zajlik, mint előző hónapban a *Berkeley* alrendszer használatakor, ezért ezt most nem részletezném. Az `eval {}` blokk utolsó utasításaként végrehajtjuk az *SQL* mondatot a megfelelő változóbehelyettesítéssel. Nagyon fontos a blokkot lezáró pontosvessző! Ezután következhet az esetleges hibák kezelése. Ha történt valamilyen gond, a `$@` változó biztosan nem üres. Ez esetben mindössze egy figyelmeztető üzenetet jelenítünk meg.

Sorozatunk utolsó állomásához érkeztünk. Remélem, sikerült bemutatnom az adatbáziskezelés alapvető fogalmait, és hogy mennyire egyszerű ezeket a gyakorlatban alkalmazni. Sok sikert kívánok a további kísérletezgetéshez, akinek pedig kérdése van, írjon bátran.

Fülöp Balázs (admin@guardware.com)
18 éves, imádja a Túró Rudit, a Debian Linuxot és a teheneket. Kedvenc írója Slawomir Mrožek. Leginkább a számítógépes hálózatok biztonsága érdekli. A BME VIK műszaki informatikus szak hallgatója.

© Kiskapu Kft. Minden jog fenntartva