Adatbázis-kezelés PHP-ben, PEAR módra

Építkezni fogunk, mégpedig PEAR modulokból. Az első górcső alá vett – és egyben legfontosabb – építőkő az adatbázis-kezelés kategóriát megalapozó DB csomag.

Sorozatunk előző epizódjában áttekintettük a PEAR rendszer alapjait, filozófiáját, megismerkedtünk az alapprogram telepítésével, majd egy egyszerű példán keresztül megismerkedhettünk egy előre elkészített modul telepítésével és használatával. A továbbiakban elsőként a méltán népszerű *PEAR DB* csomagot szeretném bemutatni, amely az egyik legöregebb, ennél fogva legjobban kiforrott elem, számos más adatbázis-kezelést végző modul épül rá, és igen jelentős felhasználói táborral rendelkezik.

A *DB* csomag fogalom a *PEAR*-t használó fejlesztők körében. Egy olyan adatbázis absztrakciós rétegről van szó, amely áttetszővé teszi a modult használók számára a rendszer alatt futó adatbázist: nem kell tudni, hogy milyen kiszolgáló fut a gépen, nem kell ismerni a sajátosságait, és adott esetben még az adatbázis-kezelőt is ki lehet cserélni a rendszer alatt, bár ez utóbbi nem szokás. A választott adatbázis-kezelő a tervezés elején meghozott stratégiai döntés, nem célszerű megváltoztatni, mert a következmények nem ismertek pontosan. Annyiból viszont rendkívül hasznos, hogy módosítás nélkül álljunk át az adott adatbázis-kiszolgáló egy újabb, némileg eltérő változatára.

A működés lényege, hogy az adatbázisunkat nem a megszokott módon, a *PHP* beépített függvényeivel kezeljük, hanem a *DB* osztály metódusainak hívásával, így fedve el előlünk a használt adatbázis-kezelőt. Ezek a metódusok egyrészt jópár feladatot átvesznek a fejlesztőtől, másrészt segítik a különböző alkalmazások, projektek egységesen történő felépítését.

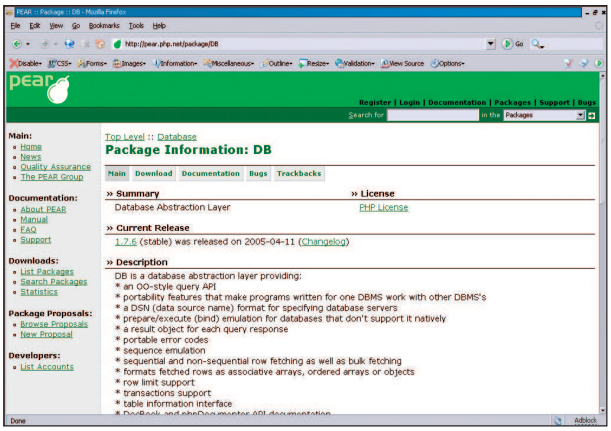
Lássunk egy konkrét példát!

Egy *mysql* adatbázisból szeretnénk egy egyszerű lekérdezés adatait kinyerni. Ehhez a következőt kell tennünk a *PHP* beépített függvényeit használva:

```
$dbh =
mysql_connect('kiszolgalo','felhasznalo','jelszo');
mysql_select_db('adatbazis_nev');
$res = mysql_query('select * from test',$dbh);
//az eredmények bejárása...
```

Mindez PEAR DB-n keresztül az alábbi módon történik:

```
$dbh =&
```



1. ábra A PEAR DB csomag weboldala

2. ábra A végfelhasználói dokumentáció példaprogramokkal

```
DB::connect("mysql://felhasznalo:jelszo@kiszolgalo/
➥adatbazis_nev");
$res =& $dbh->query("select * from test");
//az eredmenyek bejarasa...
```

(Megjegyzés: a referencia átadást végrehajtó operátorra (=&) csak *PHP4* esetében van szükség, *PHP5*-ben minden objektum referencia formájában adódik át az egyenlőség operátor hatására.)

Mint látható, minden későbbi művelet során a kapcsolatkezelő (`$dbh`) metódusait kell meghívnia *PHP* függvények helyett, ezzel egy egységes adatbázis-kezelő felületet kaptunk, amit mindenhonnan, mindenféle adatbázis-kiszolgáló mellett ugyanúgy kell használni.
Mielőtt részletes bemutató formájában megnéznénk, hogy hogyan is kell pontosan használni, tekintsük át a megoldás felépítését. Maga a csomag több osztályból áll, ennek leggyakrabban használt eleme a *DB* osztály maga, ez képviseli a felhasználók felé a *PEAR DB*-t, ezen az osztályon keresztül lehet csatlakozni, kapcsolatot bontani, lekérdezni, hibát kezelni, stb. Amikor egy kapcsolat felépül, a használat adatbázistól függően más-más módon kell az elvégzendő műveleteket leképezni. Erre szolgálnak a közvetlen adatbázis osztályok, amelyekből minden adatbázistípushoz (*MySQL*, *PostgreSQL*, *Oracle*, stb) pontosan egy tartozik. A tényleges műveletek tehát egy ilyenen keresztül zajlanak, s az értékek továbbadódnak a *DB* osztálynak, amelyik visszaadja a felhasználók felé. Ezek az osztályok a *DB* felé egyformák, mindet ugyanúgy kezeli a központi osztály, miután a kapcsolat felépítése során kiválasztotta a neki megfelelőt. Az egyformaságot úgy érik el, hogy minden ilyen adatbázis-függő közvetlen osztály egy azonos ősből, a `DB_common` alaposztályból származik, ezáltal örökli annak tagfüggvényeit. A már emlegetett *adatbázis >> függő közvetlen osztály >> DB osztály >> felhasználó* lánc legvégén (a felhasználó előtt) még van egy elem, ugyanis a felhasználó egy eredményhalmazt kap, amely itt az egyértelműség kedvéért szintén objektum, amely magában hordozza az adatok barátságos kinyeréséhez szükséges metódusokat. A visszaadott eredményhalmaz a `DBresult` *burkoló (wrapper) osztály* egy példánya.
Ezen kívül van még pár osztály, amely az alapok elsajátítása szempontjából nem lényeges, ezekre most nem térünk ki, inkább lássuk magát a medvét! Itt is meg kell jegyeznem, hogy nincs lehetőségünk az összes tagfüggvény áttekintésére, így arra fogunk törekedni, hogy egy teljes, jól áttekinthető „fejlesztői” folyamatot kapjunk kézhez a cikk végén.

Első lépés: kapcsolódás az adatbázishoz

Megszokhattuk, hogy ahányféle adatbázis, annyiféle kapcsolódási mód. A *DB* esetében ez azonban egységesítve van. Megfigyelhetjük használat közben is a fenti példakód alapján: a csatlakozáshoz egy *URL*-t definiál *(DSN – Data Source Name ~ Adatforrás név)*, amellyel leírható az adatbázis típusa, helye, felhasználónév, jelszó és az adatbázis neve, és még sok egyéb más is. A leírás kétféleképp történhet: meghatározott formátumú karakterlánccal, vagy meghatározott elemeket tartalmazó tömbbel. Sajnos adatbázis típusonként eltérő lehet a megadható komponensek neve és száma, általánosságban csak annyit lehet elmondani, hogy a *DSN* az alábbiak szerint alakul:

```
phptype(dbsyntax)://username:password@protocol+host
➥spec:port/database?option=value
```

A magyarázat:

```
phptype: az adatbázis típusa, amelyet a php kezel
dbsyntax: adatbázis „altípus” pl. ODBC vezérlő esetén
username:password: név/jelszó páros
protocol: meghatározza, hogy milyen protokollon
keresztül érjük el az adatbázist (tcp, unix socket,
stb.)
hostspec:port: Az adatbázis-kiszolgáló neve és
portja
database: a használni kívánt adatbázis
option=value&option2=value2: kulcs érték párok,
amelyekkel speciális paramétereket adthatunk meg
a kapcsolat számára.
```

A fenti példában *MySQL* adatbázis felett, „root” felhasználóként, jelszó nélkül csatlakoztunk a helyi gépen futó kiszolgálóhoz, ahol a ‘test’ nevű adatbázist használtuk.
Ha tömb formájában akarjuk megadni az adatokat, akkor asszociatív módon, kulcs-érték párokkal tehetjük meg, ahol a kulcsok a fent részletezett elemek, értékük pedig a behelyettesíteni kívánt karaktersorozatok. A kapcsolódáshoz ezután a `DB` osztály `connect()` metódusát használhatjuk, melynek első paraméter a már emlegetett *DSN*, a második pedig a kapcsolat paramétereit rögzítő tömb. Ez tartalmazza például, hogy perzisztens legyen-e a kapcsolat, használjon-e *SSL* protokollt, a nyomkövetési üzenetek szintjét, stb.
A sok elmélet után következzen egy kis gyakorlat, nézzünk egy-egy példát a karakterlánc, illetve tömb formájában átadott kapcsolódási paraméterekkel.

```
$dsn = ‘pgsql://someuser:apasswd@localhost/
➥thedb’;
$options = array(
    ‘debug’       => 2,
    ‘portability’ => DB_PORTABILITY_ALL, //minden
eszközzel támogassa az AB-függetlenséget
);

$db =& DB::connect($dsn, $options);
if (PEAR::isError($db)) {
    die($db->getMessage());
}
```

A másik esetben a kód így néz ki:

```
$dsn = array(
    ‘phptype’  => ‘pgsql’,
    ‘username’ => ‘someuser’,
    ‘password’ => ‘apasswd’,
    ‘hostspec’ => ‘localhost’,
    ‘database’ => ‘thedb’,
);
$options = array(
    ‘debug’       => 2,
    ‘portability’ => DB_PORTABILITY_ALL, //minden
eszközzel támogassa az AB-függetlenséget
);

$db =& DB::connect($dsn, $options);
if (PEAR::isError($db)) {
    die($db->getMessage());
}
```

Bizonyos esetekben néhány extra paraméter is belekerül a tömbbe, ilyen a fájl alapú ***SQLite*** adatbázis-kezelő, ahol a `mode` paraméterrel adható meg a fájl oktális jogosultsága (például `644`). Ezekről a ***DB*** csomag leírásában tájékozódhatunk.

Második lépés: a lekérdezés indítása

Ha ezzel megvolnánk, sikerült kapcsolatot teremteni a kiszolgálóval, jöhet az adatok lekérése. Ez alapesetben rendkívül egyszerű, a `DB` osztály `query()` tagfüggvényének segítségével paraméterként átadjuk az ***SQL*** lekérdezést tartalmazó karakterláncot. Az eredményt egy `DBresult` típusú objektum képében kapjuk meg.

```
include("DB.php");

$db = DB::connect("mysql://root@localhost/test");
$res = $db->query("select * from proba");
```

A dolog némiképp bonyolódik, ha szeretnénk paraméteres lekérdezéseket készíteni. Ebben az esetben a lekérdezési karakterláncban nem szerepelnek konkrét értékek, hanem azt a lekérdezőfüggvény egy másik paramétereként adjuk át.

```
$res = $db->query("select * from proba where ertek
➥= ?", 26);
```

Ekkor a kérdőjel helyére a 26-os szám helyettesítődik be. Több paraméter esetén egy tömbként kell átadni az értékeket, ahol a tömb elemei az egyes paramétereket jelentik sorrendben balról jobbra.

```
$res = $db->query("select * from proba where kulcs
➥= ? and ertek = ?", array(13,26));
```

Harmadik lépés: az eredmények kezelése

A lekérdezést legtöbb esetben azért adjuk ki, hogy feldolgozzuk annak eredményét. Szó volt róla, hogy az eredményt egy `DBresult` típusú objektum képében kapjuk meg, amely magában hordozza a feldolgozáshoz, adatkinyeréshez szükséges tagfüggvényeket.
Menjünk végig egy eredményhalmazon, és írassuk ki az egyes sorokat:

```
include("DB.php");
$db = DB::connect("mysql://root@localhost/test");
$res = $db->query("select * from proba");
while ($res->fetchInto($row, DB_FETCHMODE_ASSOC)) {
    var_dump($row);
}
```

A `fetchInto()` metódus második paramétere azt határozza meg, hogy milyen formában adja vissza az adathalmaz következő sorát. Alapértelmezetten a `DB_FETCHMODE_ORDERED` konstans van érvényben, amely egy számokkal indexelt tömbben tartalmazza sorban az egyes mezők tartalmát. A kódrészletben is alkalmazott `DB_FETCHMODE_ASSOC` asszociatív tömbbel tér vissza, ahol a kulcsok a mező nevei, az értékek a mező értékei. A harmadik lehetőség a `DB_FETCHMODE_OBJECT` konstans használata, amelynek hatására a visszatérési érték egy objektum lesz, ahol az osztályváltozók neve a mezőnevekkel egyezik meg, az értéke pedig a mezők tartalmával.
Az eredményhalmazból egyéb értékes információ is kideríthető. A `numRows()` metódus visszaadja, hogy hány sorral tért vissza a lekérdezés, a `numCols()` ehhez hasonlóan azt, hogy hány oszlopa van az eredményhalmaznak. Ezen kívül lekérdezhetjük az halmaz részletes felépítését a `tableInfo()` tagfüggvény segítségével, amely tartalmazza, hogy milyen oszlopai vannak az eredménynek, és azok milyen típusúak, milyen méretűek.

Negyedik lépés: lekérdezés indítása előkészítéssel

Számtalan esetben előállhat az a helyzet, hogy hasonló módosító jellegű lekérdezéseket kell futtatnunk, ahol kizárólag a módosított értékek változnak, a lekérdezés többi része ugyanaz marad. Erre számtalan adatbázis-kezelő alkalmazza azt a megoldást, hogy paraméteres lekérdezéseket használva első lépésben előkészíti, „lefordítja" a lekérdezést, majd ezt az előkészített parancsot tudjuk később, akár egymás után sokszor lefuttatni.
Például nézzük azt az esetet, amikor minden alkalmazott fizetését meg kell növelni egy előre meghatározott összeggel, amit a vezetőség állított elő személyre szólóan. Az ***SQL*** parancs az alábbiak szerint néz ki:

```
'update employees set salary=salary + ? where
➥id = ?'
```

A fentiekből már tudjuk, hogy a lekérdezés indítása során helyettesíthetjük be a ? jelek helyére a kívánt értékeket. A lekérdezést előkészítve, majd az előkészített parancsot mindig a megfelelő paraméterrel elindítva egyetlen fordítással megoldható a művelet.
A ***PEAR DB*** biztosítja számunkra ezt a lehetőséget. A példát lefordítva az alábbi kódrészletet kell végrehajtanunk:

```
include("DB.php");
$db = DB::connect("mysql://root@localhost/test");
$qry='update employees set salary=salary + ? where
➥id = ?';
$sth=$db->prepare($qry);
foreach ($salary_lift as $id=>$value) {
        $db->execute($sth,array($id,$value));
}
```

Jól látható, hogy egy referenciát kapunk vissza az előkészített lekérdezésre, amelyet aztán ugyanúgy kell elindítani, mintha lekérdezés lenne, még a paramétereket is ugyanabban a tömbös formában kell átadni, ha több paramétert szeretnénk használni. Szót kell ejtenünk a lekérdezésben használatos helyfenntartókat (placeholder), ami az eddigiekben csak a ? jel volt. Ezeknek a helyére helyettesítődnek be a második paraméterként átadott változók.
PEAR DB terminológiával a ? jel skalár értékek helyét jelöli. A behelyettesítés során automatikusan átalakítja a speciális karaktereket (escape), illetve ha karaktersorozatról van szó, aposztrofok közé teszi a ***DB*** a használt adabázis-kezelő elvárásainak megfelelően.

A ! jel működése megegyezik az előzővel, azt leszámítva, hogy ott az értékek úgy helyettesítődnek be, ahogy átadtuk, tehát nincs speciális karakterek átalakítása és aposztrofok közé tétel.
Az utolsó az & jel, amely egy valódi fájlnevet vár a behelyettesítési folyamat során. A behelyettesített érték ekkor a megadott fájl tartalma lesz. Ez olyan esetekben használható jól, ahol fájlokat, vagy más bináris adatokat szeretnénk az adatbázisba tölteni.
Felmerülhet sokunkban a kérdés a fenti példa alapján: ha egyszer úgyis tömbben vannak az adatok, és a megoldást pont arra találták ki, hogy ismétlődő szerkezetű adatokat minél gyorsabban lehessen adatbázisba tenni, akkor mi a fenének bejárni a tömböt, és egyenként betenni az értékeket? A válasz az, hogy nem kell, létezik erre is megoldás, amelyet az `executeMultiple()` tagfüggvény valósít meg.

```
include("DB.php");
$db = DB::connect("mysql://root@localhost/test");
$qry='update employees set salary=salary + ? where
➥id = ?';
$sth=$db->prepare($qry);
$db->execute($sth,$salary_lift);
```

Az előkészítés-indítás jelleggel alkotott lekérdezést nem támogatja minden adatbázis-kezelő, de mi bárhol használhatjuk, mivel a `DB` alapértelmezetten emulálja ezeket a funkciókat számunkra (némi sebességcsökkenés árán természetesen).

Ötödik lépés: hibakezelés

A PEAR statikus függvényt kínál az eredményhalmazok, adatbázis leírók ellenőrzésére. Ha valamely művelet során hiba történt, akkor az adott művelet visszatérési értéke egy `DB_Error` típusú objektum, amely ugyanazokat a tagfüggvényeket kínálja, mint a `PEAR_Error` osztály. Ha tehát hibát szeretnénk észlelni, akkor a hiba elkövetése után a művelet során keletkezett eredményhalmazt kell vizsgálat tárgyává tenni, mégpedig a `PEAR` (ill. `DB_Error`) osztály statikus `isError()` tagfüggvényének segítségével.

Kapcsolódási hiba észlelése

```
$db = DB::connect("mysql://root@localhost/test");
if (PEAR::isError($db)) {
        die('Kapcsolódás sikertelen');
}
```

Lekérdezés futtatási hiba észlelése

```
include("DB.php");
$db = DB::connect("mysql://root@localhost/test");
$qry='update employees set salary=salary + ? where
➥id = ?';
$sth=$db->prepare($qry);
$res=$db->execute($sth,$salary_lift);
if (PEAR::isError($res)) {
        $res->getMessage();     //visszaadja
                                //a hibaüzenete
        $res->getCode();        //visszaadja
                                //a hibakódot
}
```

Mint látható, a `DB_Error` osztály is magában hordozza a hibakezelésre vonatkozó összes tagfüggvényt.

A PEAR DB előnyei

A fentiek alapján nem kell sokat ecsetelni a megoldás előnyeit: egyszerű, kis túlzással szabványos is, széles körben elterjedt, támogatott, dokumentált és nem nekünk kell megírni. Ez utóbbi két szempont igen jelentős: nem csak a kód megírására fordítandó időt takarítjuk meg, de a programhibák száma is minimális. Ezen túl már említettem, hogy ha más fejlesztő nyúl a kódhoz, nagy valószínűséggel ismerni fogja. Ha azonban ez nincs is így, a csomag igen jól dokumentált: nem csak ***API*** referencia áll rendelkezésre, de a ***PEAR*** honlapján részletes végfelhasználói dokumentációt találunk szemléletes példákkal illusztrálva.
A ***DB*** megoldásai rendkívül hasonlítanak például a ***JDBC*** filozófiához, ahol szintén minden adatbázist ugyanúgy kell kezelni: objektum-orientált és átgondoltságot sugároz, így nem csak ***PEAR DB*** fejlesztők tudják hatékonyan elsajátítani a használat rejtelmeit, de olyanok is, akik ***JDBC*** adatbázis absztrakciós rétegen nőttek fel.

A PEAR DB hátrányai

A modul rendkívül általános, számtalan dolgot tud, ennélfogva igen valószínű, hogy számtalan szolgáltatását nem is fogjuk soha igénybe venni. Ezek csak ott vannak, lassítják a kódot, stb. Másfelől a valódi funkciók elrejtése (előkészítéses lekérdezés nem támogatott adatbázison, eredményhalmazok burkolása, stb). jelentős többletterhelést okoz a programnak. Ez általában igaz minden emelt szintű felületre is. A dolog legteteje pedig az, hogy a kódunk hordozható marad az adatbázis-kiszolgáló fölött, annak típusától függetlenül. Az elején már említettem, hogy ezt a lehetőséget a legritkább esetben alkalmazzuk. Igaz, lehet a kompatibilitás mértékét szabályozni a kapcsolat felépítésekor, de még így is számtalan felesleges menet közbeni átalakítást cipel a hátán a rendszer.
Mindezek ellenére személy szerint egyértelműen javaslom a használatát! A hétköznapi alkalmazások esetében soha nincs olyan teljesítményigény, amely kizárttá tenné a ***DB*** alkalmazásának lehetőségét, viszont szép, áttekinthető, átgondolt felépítésű kóddal büszkélkedhetünk.
A ***DB*** csomagról igen részletes leírást talál az érdeklődő olvasó a ***http://www.pear.php.net/manual/en/package.database.db.php*** címen. Ez a végfelhasználói dokumentáció sok-sok kódrészlettel.
Aki ezzel nem elégszik meg, és ki akarja vesézni a teljes csomagot, annak ajánlom a ***http://www.pear.php.net/package/DB/docs/latest/*** címen elérhető ***API*** referencia leírást. Kevésbé érthető, mint az előző, viszont tényleg minden aprólékos információ megtalálható benne.

Komáromi Zoltán

KAPCSOLÓDÓ CÍMEK

A PEAR DB elérhetősége:
➲ http://www.pear.php.net/package/DB/