Desenvolupament d'aplicacions en connexió amb bases de dades

Marc Gibert Ginestà

P06/M2009/02153

UOC
www.uoc.edu

Índex

Introducció 5

Objectius 6

1. Connexió i ús de bases de dades en llenguatge PHP 7
- 1.1. API nativa enfront d'API amb abstracció 7
- 1.2. API nativa en MySQL 8
- 1.3. API nativa en PostgreSQL 12
- 1.4. Capa d'abstracció PEAR::DB 17
  - 1.4.1. Capa d'abstracció del motor de la base de dades 19
  - 1.4.2. Transaccions 24
  - 1.4.3. Seqüències 24

2. Connexió i ús de bases de dades en llenguatge Java 27
- 2.1. Accedir a l'SGBD amb JDBC 28
- 2.2. Sentències preparades 31
- 2.3. Transaccions 32

Resum 34

Bibliografia 35

Introducció

Un curs de bases de dades quedaria incomplet si únicament veiéssim el funcionament i l'administració dels dos gestors comentats anteriorment. Un dels principals objectius d'un SGBD és proporcionar un sistema d'emmagatzemament i consulta de dades a què es puguin connectar les aplicacions que desenvolupem.

Així doncs, en aquest capítol abordarem aquest tema des d'una perspectiva totalment pràctica, intentant exposar les bases per a usar els SGBD vistos anteriorment des d'alguns dels llenguatges de programació i connectors més usats. Els exemples proporcionats seran tan simples com sigui possible per a centrarnos en el tema que ens ocupa i no en les particularitats del llenguatge de programació per si mateix.

En primer lloc, veurem les eines que ofereix PHP per connectar-se amb bases de dades, i en proporcionarem alguns exemples.

A continuació, passarem a examinar la connexió JDBC a SGBD en general i a MySQL i PostgreSQL en particular, proporcionant-ne també els exemples necessaris. També comentarem algun aspecte avançat com el de la persistència de la connexió a l'SGBD.

Objectius

L'objectiu principal d'aquesta unitat és conèixer les diferents tècniques de connexió a bases de dades que ofereixen PHP i Java.

Més concretament, els objectius que hauríeu d'assolir en acabar el treball amb aquesta unitat són els següents:

- Conèixer les possibilitats que PHP i Java ofereixen per a la connexió i ús de bases de dades en general, i de MySQL i PostgreSQL en particular.

- Saber adaptar els programes desenvolupats en aquests llenguatges perquè utilitzin SGBD.

1. Connexió i ús de bases de dades en llenguatge PHP

El llenguatge de *script* PHP s'ha popularitzat extraordinàriament durant els últims anys, gràcies a la seva senzillesa i la seva sintaxi heretada d'altres llenguatges com C, Perl o Visual Basic, que gairebé tots els desenvolupadors ja coneixien en major o menor grau.

La seva fàcil integració amb els servidors web més populars (Apache, IIS, etc.), sense necessitat de recompilacions o configuracions complexes, ha contribuït també al fet que gairebé tots els proveïdors d'espai web, desenvolupadors d'aplicacions de programari lliure basades en web i proveïdors d'aplicacions empresarials l'usin per als seus productes.

Al llarg de la seva curta història ha progressat significativament, i la versió 5.0 representa un pas endavant en l'orientació a objectes, el tractament d'excepcions i el treball amb XML, la qual cosa fa que s'assembli en prestacions als llenguatges més madurs en l'àmbit empresarial.

> La versió 5.0 era la més actualitzada en el moment de confecció d'aquest material (final del 2004). En aquest capítol, tanmateix, treballarem amb la versió 4, ja que la versió 5.0 és molt recent i els canvis que incorpora quant a configuració, model de programació, etc. no seran familiars a la majoria dels estudiants amb coneixements de PHP.

1.1. API nativa enfront d'API amb abstracció

Des de la versió 2.0, PHP ha incorporat de manera nativa funcions per a la connexió i l'ús de bases de dades. En ser la rapidesa una de les màximes d'aquest llenguatge, i davant de l'avantatge que proporciona mecanismes per a la càrrega de llibreries externes, es van crear unes llibreries per a cada motor de base de dades, que contenien les funcions necessàries per a treballar-hi.

Aquestes API natives són diferents per a cada SGBD, tant en els noms de les funcions que s'utilitzen per a crear una connexió a la base de dades, llançar una consulta, etc., com en el tractament d'errors, resultats, etc.

Tot i que es pot argumentar que en usar l'API de l'SGBD concret que utilitzem disposarem d'operadors o funcionalitats específiques d'aquest motor que una llibreria estàndard no pot proporcionar, amb el pas del temps s'ha vist que la utilització d'aquestes API només està indicada (i tot i així, no és recomanable) per a aplicacions que sapiguem segur que no canviaran l'SGBD amb què tre-

> Com que actualment hi ha aplicacions web desenvolupades en PHP que usa l'API concreta de l'SGBD per al qual van ser pensades, les revisarem en aquest apartat.

ballen, ja que la revisió del codi PHP, quan hi ha un canvi d'SGBD, és molt costosa i procliu a errors.

1.2. API nativa en MySQL

Per a treballar amb l'API nativa de MySQL en PHP, haurem d'haver compilat l'intèrpret amb suport per a aquest SGBD, o bé disposar ja del binari de PHP precompilat amb el suport incorporat.

En cas d'haver-lo de compilar, únicament haurem d'indicar com a opció `--with-mysql`. Posteriorment, o en cas que ja disposem del binari, podem validar que el suport per a MySQL està inclòs correctament en l'intèrpret amb l'execució de l'ordre següent:

```
$ php -i | grep MySQL
supported databases => MySQL
MySQL Support => enabled
$
```

A partir d'aquí, PHP proporciona uns paràmetres de configuració que ens permetran controlar alguns aspectes del funcionament de les connexions amb l'SGBD, i les mateixes funcions de treball amb la base de dades.

Quant als paràmetres, s'hauran de situar en el fitxer `php.ini`, o bé configurar-se per a la nostra aplicació en concret des del servidor web. Destaquen els següents:

- `mysql.allow_persistent`: indica si permetrem connexions persistents a MySQL. Els valors possibles són `true` o `false`.
- `mysql.max_persistent`: nombre màxim de connexions persistents permeses per procés.
- `mysql.max_links`: nombre màxim de connexions permeses per procés, incloent les persistents.
- `mysql.connect_timeout`: temps que ha de transcórrer, en segons, abans que PHP abandoni l'intent de connexió al servidor.

Les connexions persistents són connexions a la base de dades que es mantenen obertes per a evitar el temps de latència que es perd en connectar i desconnectar. L'intèrpret, en executar la sentència de connexió a la base de dades, examina si hi ha cap altra connexió oberta sense usar, i torna aquesta en lloc d'obrir-ne una de nova. El mateix es produeix en desconnectar, l'intèrpret pot fer la desconnexió si hi ha prou connexions encara obertes, o bé mantenir la connexió oberta per a futures consultes.

Pel que fa a la utilització de l'API per a la connexió i consulta de bases de dades, començarem amb un exemple:

```
1  <?php
2  // Connectant i triant la base de dades amb què treballarem
3  $link = mysql_connect('host_mysql', 'usuari_mysql', 'password_mysql') or die
('No puc connectar-me: ' . mysql_error());
4  echo 'Connexió establerta';
5  mysql_select_db('lameva_database',$link) or die('No he pogut accedir a la base
de dades');
6
7  // Fent una consulta SQL
8  $query = 'SELECT * FROM lameva_taula';
9  $result = mysql_query($query,$link) or die('Consulta errònia: ' . mysql_error());
10
11 // Mostrem els resultats en HTML
12 echo "<table>\n";
13 while ($line = mysql_fetch_array($result, MYSQL_ASSOC)) {
14   echo "\t<tr>\n";
15   foreach ($line as $col_value) {
16   echo "\t\t<td>$col_value</td>\n";
17   }
18 echo "\t</tr>\n";
19 }
20 echo "</table>\n";
21
22 // Alliberem el resultset
23 mysql_free_result($result);
24
25 // Tanquem la connexió
26 mysql_close($link);
27 ?>
```

En les cinc primeres línies s'estableix la connexió i se selecciona la base de dades amb què es treballarà. El codi és bastant explícit i la majoria d'errors se solen deure a una mala configuració dels permisos de l'usuari sobre la base de dades amb què ha de treballar.

Convé estar molt atent, sobretot a les adreces d'origen de la connexió, ja que, encara que podem usar `localhost` com a nom d'equip, si l'intèrpret i l'SGBD són al mateix servidor, sol ocórrer que PHP resol `localhost` al nom real de l'equip i intenta connectar-se amb aquesta identificació. Així doncs, hem d'examinar amb cura els arxius de registre de MySQL i els usuaris i privilegis d'aquest si falla la connexió.

Per a establir una connexió persistent, usarem la funció `mysql_pconnect()` amb els mateixos paràmetres.

A continuació, s'utilitza la funció `mysql_query()` per a llançar la consulta en la base de dades.

Utilitat de la sentència de connexió

Hem proporcionat la sentència SQL a la funció i l'enllaç ens ha tornat la sentència de connexió. Aquesta funcionalitat és absolutament necessària si es treballa amb diverses connexions simultàniament, si únicament hi ha una connexió establerta en l'*script*, aquest paràmetre és opcional.

La funció `mysql_error()` és universal, i torna l'últim error ocorregut en l'SGBD amb la nostra connexió, o amb la connexió `$link` que li indiquem com a paràmetre.

La funció `mysql_query()` pot tornar els resultats següents:

- FALSE si hi ha hagut un error.
- Una referència a una estructura si la sentència és de consulta i ha tingut èxit.
- TRUE si la sentència és d'actualització, esborrament o inserció i ha tingut èxit.

La funció `mysql_affected_rows()` ens permet conèixer el nombre de files que s'han vist afectades per sentències d'actualització, esborrament o inserció.

La funció `mysql_num_rows()` ens permet conèixer el nombre de files tornat per sentències de consulta.

Una vegada obtingut el recurs a partir dels resultats de la consulta, PHP proporciona un gran nombre de maneres d'iterar sobre els seus resultats o d'accedir a un d'ells directament. Comentem les més destacades:

- `$fila = mysql_fetch_array($recurs, < tipus_d_array >)`

  Aquesta funció va iterant sobre el recurs, tornant una fila cada vegada, fins que no en queden més, i torna FALSE. La forma de l'*array* tornat dependrà del paràmetre `< tipus_d_array >` que pot prendre aquests valors:

– MYSQL_NUM: torna un *array* amb índexs numèrics per als camps. És a dir, en `$fila[0]` tindrem el primer camp del SELECT, en `$fila[1]`, el segon, etc.

– MYSQL_ASSOC: torna un *array* associatiu en el qual els índexs són els noms de camp o àlies que hàgim indicat en la sentència SQL.

– MYSQL_BOTH: torna un *array* amb els dos mètodes d'accés.

Recuperant l'exemple inicial, entre les línies 7 i 21:

```
7  // Realitzant una consulta SQL
8  $query = 'SELECT * FROM lameva_taula';
9  $result = mysql_query($query,$link) or die('Consulta errònia: ' . mysql_error());
10
11 // Mostrem els resultats en HTML
12 echo "<table>\n";
13 while ($line = mysql_fetch_array($result, MYSQL_BOTH)) {
14   echo "\t<tr>\n";
15   for($i=0;$i<sizeof($line);$i++) {
16     echo "\t\t<td>$line[$i]</td>\n";
17   }
18   echo "<td>Nom: $line['nom']</td>";
19   echo "\t</tr>\n";
20 }
21 echo "</table>\n";
```

- `$objecte= mysql_fetch_object($recurs)`

  Aquesta funció va iterant sobre els resultats, tornant un objecte cada vegada, de manera que l'accés a les dades de cada camp es fa per mitjà de les propietats de l'objecte. Igual que en l'*array* associatiu, cal vigilar amb els noms dels camps en consulta per a evitar que torni camps amb el mateix nom fruit de combinacions de diverses taules, ja que només podrem accedir a l'últim d'aquests:

  Tornem sobre l'exemple inicial

```
7  // Realitzant una consulta SQL
8  $query = 'SELECT nom,cognoms FROM lameva_taula';
9  $result = mysql_query($query,$link) or die('Consulta errònia: ' . mysql_error());
10
11 // Mostrem els resultats en HTML
12 echo "<table>\n";
13 while ($object = mysql_fetch_array($result, MYSQL_BOTH)) {
14   echo "\t<tr>\n";
15   echo "<td>Nom: " . $object->nom . "</td>";
16   echo "<td>Cognoms: " . $object->cognoms . "</td>";
17   echo "\t</tr>\n";
18 }
19 echo "</table>\n";
```

- `$valor = mysql_result($recurs,$numero_de_fila,$numero_de_camp)`

  Aquesta funció consulta directament un valor d'un camp d'una fila especificada. Pot ser útil si volem conèixer un resultat sense necessitat de fer bucles innecessaris. Evidentment, hem de saber exactament on es troba.

- `$exit = mysql_data_seek($recurs,$fila)`

  Aquesta funció permet moure el punter dins del full de resultats representat per `$recurs`, fins a la fila que vulguem. Pot ser útil per a avançar o per

a retrocedir i tornar a recórrer el full de resultats sense haver d'executar la sentència SQL de nou. Si la fila sol·licitada no existeix al full de resultats, el resultat serà FALSE, igual que si el full no conté cap resultat. És a dir, el valor de `$fila` ha de ser entre 0 (la primera fila) i `mysql_num_rows()-1`, excepte quan no hi ha cap resultat, cas en què tornarà FALSE.

Finalment, comentarem les funcions d'alliberament i desconnexió. En el primer cas, PHP fa una feina excel·lent alliberant recursos de memòria quan l'execució en curs ja no els utilitzarà més. Tot i així, si la consulta torna un full de dades molt gran, pot ser convenient alliberar el recurs quan no el necessitem.

Pel que fa al tancament de la connexió, tampoc no sol ser necessari, ja que PHP tanca totes les connexions en finalitzar l'execució i, a més, el tancament sempre està condicionat a la configuració de les connexions persistents. Tal com ja hem comentat, si activem les connexions persistents (o bé hem connectat amb `mysql_pconnect`), aquesta funció no té cap efecte i, en tot cas, serà PHP qui decideixi quan es tancarà cada connexió.

Ja hem comentat que les API específiques per a cada motor inclouen un conjunt de funcions que podia ajudar a treballar amb els seus aspectes particulars, a continuació, enumerem les més importants:

- `mysql_field_flags`, `mysql_field_name`, `mysql_field_table`, `mysql_field_type`: aquestes funcions reben com a paràmetre un `$recurs` i un índex de camp dins de la consulta executada i tornen informació sobre el camp; en concret, les seves restriccions, nom, taula a què corresponen i tipus de camp. Poden ser molt útils per a treballar amb consultes genèriques sobre bases de dades i/o camps que no coneixem en realitzar l'*script*.

- `mysql_insert_id`: aquesta funció torna l'últim identificador obtingut d'una inserció en un camp autoincremental.

- `mysql_list_dbs`, `mysql_list_tables`, `mysql_list_fields`: amb diferents paràmetres, aquestes funcions permeten consultar dades d'administració del motor de la base de dades.

Bibliografia

Hi ha altres funcions més específiques per a obtenir informació sobre el client que origina la connexió, o sobre el mateix servidor on s'està executant. Convé consultar la documentació per a obtenir informació sobre usos més avançats d'aquesta API.

1.3. API nativa en PostgreSQL

Per a treballar amb l'API nativa de PostgreSQL en PHP, haurem d'haver compilat l'intèrpret amb suport per a aquest SGBD, o bé disposar ja del binari de PHP precompilat amb el suport incorporat.

En cas d'haver-lo de compilar, únicament hem d'indicar com a opció `--with-pgsql`. Posteriorment, o en cas que ja disposem del binari, podem

validar que el suport per a PostgreSQL està inclòs correctament en l'intèrpret amb l'execució de l'ordre següent:

```
$ php -i | grep PostgreSQL
PostgreSQL
PostgreSQL Support => enabled
PostgreSQL(libpq) Versio => 7.4.6
$
```

A partir d'aquí, PHP proporciona uns paràmetres de configuració que ens permetran controlar alguns aspectes del funcionament de les connexions amb l'SGBD, i les mateixes funcions de treball amb la base de dades.

Quant als paràmetres, s'hauran de situar en el fitxer `php.ini` o bé configurar-se per a la nostra aplicació en concret des del servidor web. Destaquen els següents:

- `pgsql.allow_persistent`: indica si permetrem l'ús de connexions persistents. Els valors són `true` o `false`.
- `pgsql.max_persistent`: nombre màxim de connexions persistents permeses per procés.
- `pgsql.max_links`: nombre màxim de connexions permeses per procés, incloent-hi les persistents.
- `pgsql.auto_reset_persistent`: detecta automàticament connexions persistents tancades i les elimina.

Aquest paràmetre disminueix lleugerament el rendiment del sistema.

Pel que fa a la utilització de l'API per a la connexió i consulta de bases de dades, reproduirem l'exemple anterior:

```
1  <?php
2  // Connectant i triant la base de dades amb què treballarem
3  $link = pg_connect("host=host_pgsql port=5432 dbname=mi_database user=user_pgsql
password=pass_pgsql");
4  $stat = pg_connection_status($link);
5  if ($stat === 0) {
6     echo 'Connexió establerta';
```

```
7  } else {
8    die 'No puc connectar-me';
9  }
10
11 // Realitzant una consulta SQL
12 $query = 'SELECT * FROM lameva_taula';
13 $result = pg_query($link,$query) or die('Consulta errònia: ' . pg_last_error());
14
15 // Mostrem els resultats en HTML
16 echo "<table>\n";
17 while ($line = pg_fetch_array($result, PGSQL_ASSOC)) {
18    echo "\t<tr>\n";
19    foreach ($line as $col_value) {
20    echo "\t\t<td>$col_value</td>\n";
21    }
22 echo "\t</tr>\n";
23 }
24 echo "</table>\n";
25
26 // Alliberem el resultset
27 pg_free_result($result);
28
29 // Tanquem la connexió
30 pg_close($link);
31 ?>
```

En les deu primeres línies, establim la connexió i comprovem que s'ha realitzat correctament.

A diferència de MySQL, la selecció de la base de dades es fa en el moment de la connexió. En canvi, la comprovació de la connexió és una mica més complicada.

Per a establir una connexió persistent, usarem la funció `pg_pconnect()` amb els mateixos paràmetres.

A continuació, s'utilitza la funció `pg_query()` per a llançar la consulta a la base de dades.

Per a comprovar errors, l'API de PostgreSQL distingeix entre un error de connexió, i errors sobre els recursos tornats. En el primer cas, haurem d'usar `pg_connection_status`(); en el segon podem optar per `pg_last_error()` o bé `pg_result_error($recurs)` per a obtenir el missatge d'error que pugui haver tornat un recurs en concret.

Recordeu

La majoria dels errors en aquest aspecte s'originen per una mala configuració dels permisos dels usuaris en l'SGBD. Una manera de provar-ho que sol oferir més informació és intentar la connexió amb el client de la base de dades des de la línia d'ordres, especificant el mateix usuari, base de dades i contrasenya que utilitzem en l'*script*.

Utilitat de la sentència de connexió

Aquí s'apliquen els mateixos consells que donàvem en l'apartat anterior, i les mateixes consideracions quant al paràmetre `$link` i la seva opcionalitat si únicament tenim una connexió establerta amb el mateix usuari i contrasenya.

La funció pg_query() pot tornar els resultats següents:

- FALSE si hi ha hagut un error.
- Una referència a una estructura si la sentència ha tingut èxit.

La funció `pg_affected_rows($recurs)` ens permet conèixer el nombre de files que s'han vist afectades per sentències d'actualització, esborrament o inserció. Aquesta funció haurà de rebre com a paràmetre el recurs tornat per la funció `pg_query()`.

La funció `pg_num_rows($recurs)` ens permet conèixer el nombre de files tornat per sentències de consulta.

Una vegada obtingut el recurs a partir dels resultats de la consulta, PHP proporciona un gran nombre de maneres d'iterar sobre els seus resultats o d'accedir a un d'aquests directament. Comentem les més destacades:

- `$fila = pg_fetch_array($recurs, < tipus_d_array >)`

  Aquesta funció va iterant sobre el recurs, tornant una fila cada vegada, fins que no en queden més, i torna FALSE. La forma de l'*array* tornat dependrà del paràmetre `< tipus_d_array >` que pot prendre aquests valors:

- PG_NUM: torna un *array* amb índexs numèrics per als camps. És a dir, en `$fila[0]` tindrem el primer camp del SELECT, en `$fila[1]`, el segon, etc.

- PG_ASSOC: torna un *array* associatiu on els índexs són els noms de camp o àlies que hàgim indicat en la sentència SQL.

- PG_BOTH: torna un *array* amb els dos mètodes d'accés.

  Recuperant l'exemple inicial, entre les línies 11 i 24:

```
11 // Realitzant una consulta SQL
12 $query = 'SELECT * FROM mi_tabla';
13 $result = pg_query($query,$link) or die('Consulta errònia: ' . pg_last_error());
14// Mostrem los resultats en HTML
15 echo "<table>\n";
16 while ($line = pg_fetch_array($result, PGSQL_BOTH)) {
17   echo "\t<tr>\n";
18   for($i=0;$i<sizeof($line);$i++) {
19     echo "\t\t<td>$line[$i]</td>\n";
20   }
21   echo "<td>Nom: $line['nom']</td>";
22   echo "\t</tr>\n";
23 }
24 echo "</table>\n";
```

- `$objecte = pg_fetch_object($recurs)`

  Aquesta funció va iterant sobre els resultats, tornant un objecte cada vegada, de manera que l'accés a les dades de cada camp es realitza per mitjà de les propietats de l'objecte. Igual que en l'*array* associatiu, cal vigilar amb els noms dels camps en consulta i evitar que torni camps amb el mateix nom fruit de combinacions de diverses taules, ja que només podrem accedir a l'últim d'aquests.

  Tornem sobre l'exemple inicial

```
11 // Realitzant una consulta SQL
12 $query = 'SELECT nom,cognoms FROM lameva_taula';
13 $result = pg_query($query,$link) or die('Consulta errònia: ' . pg_last_error());
14 // Mostrem els resultats en HTML
15 echo "<table>\n";
16 while ($object = pg_fetch_array($result, PGSQL_BOTH)) {
17   echo "\t<tr>\n";
18   echo "<td>Nom: " . $object->nom . "</td>";
19   echo "<td>Cognoms: " . $object->cognoms . "</td>";
20   echo "\t</tr>\n";
21 }
22 echo "</table>\n";
```

  Podem passar a la funció `pg_fetch_object()` un segon paràmetre per a indicar la fila concreta que volem obtenir:

  `$resultat = pg_fetch_all($recurs)`

  Aquesta funció torna tot el full de dades corresponent a `$recurs`; és a dir, un *array* amb totes les files i columnes que formen el resultat de la consulta.

- `$exit = pg_result_seek($recurs,$fila)`

  Aquesta funció permet moure el punter dins del full de resultats representat per `$recurs` fins a la fila que vulguem. S'han de prendre les mateixes consideracions que en la funció `mysql_data_seek()`.

Quant a l'alliberament de recursos i la desconnexió de la base de dades, és totalment aplicable el que s'ha explicat per a MySQL, incloent els aspectes relacionats amb les connexions persistents.

Igual que en MySQL, PHP també proporciona funcions específiques per a treballar amb alguns aspectes particulars de PostgreSQL. En tenir aquest més funcionalitat

que s'allunya de l'estàndard a causa del seu suport a objectes, aquestes funcions cobraran més importància. A continuació comentem les més destacades:

- `pg_field_name`, `pg_field_num`, `pg_field_size`, `pg_field_type`: aquestes funcions proporcionen informació sobre els camps que integren una consulta. Els seus noms són prou explícits sobre la seva comesa.

- `pg_last_oid`: aquesta funció ens torna l'OID obtingut per la inserció d'una tupla si el recurs que rep com a paràmetre és el corresponent a una sentència INSERT. En cas contrari torna FALSE.

- `pg_lo_create`, `pg_lo_open`, `pg_lo_export`, `pg_lo_import`, `pg_lo_read`, `pg_lo_write`: aquestes funcions (entre d'altres) faciliten el treball amb objectes grans (LOB) en PostgreSQL.

```
<?php
  $database = pg_connect("dbname=jacarta");
  pg_query($database, "begin");
  $oid = pg_lo_create($database);
  echo "$oid\n";
  $handle = pg_lo_open($database, $oid, "w");
  echo "$handle\n";
  pg_lo_write($handle, "large object data");
  pg_lo_close($handle);
  pg_query($database, "commit");
?>
```

Les funcions `pg_lo_import` i `pg_lo_export` poden prendre fitxers com a paràmetres, facilitant la inserció d'objectes binaris en la base de dades.

Nota

Hi ha altres funcions que tenen la mateixa comesa que combinacions d'algunes de les funcions comentades anteriorment (per exemple, `pg_select` o `pg_insert`, `pg_copy_from`), però que no es comenten en aquest material per la seva extensió i pel seu poc ús.

1.4. Capa d'abstracció PEAR::DB

El PEAR (PHP *extension and application repository*) es defineix com un marc de treball i un sistema de distribució de llibreries reutilitzables per a PHP. És similar en concepte al CPAN (*comprehensive perl archive network*) del llenguatge Perl o al PyPI (*Python package index*) de Python.

El PEAR pretén proporcionar una llibreria estructurada de codi i llibreries reutilitzables, mantenir un sistema per proporcionar a la comunitat eines per a compartir els seus desenvolupaments i fomentar un estil de codificació estàndard en PHP.

PEAR

Si la nostra instal·lació de PHP és recent, ja disposarem de PEAR instal·lat (tret que l'hàgim compilat amb l'opció `--without-pear`.

PEAR ha de ser el primer recurs per a resoldre qualsevol carència detectada en les funcions natives de PHP. Com a bona pràctica general en el món del programari lliure, sempre és millor usar, aprendre o millorar a partir del que han

fet d'altres que proposar-nos de reinventar la roda. A més, si fem millores en les llibreries que usem de PEAR, sempre podem contribuir a aquests canvis mitjançant les eines que ens proporciona.

En certa manera, PEAR es comporta com un gestor de paquets més del que poden incorporar les distribucions GNU/Linux més recents (com apt, yum o YOU). Aquest gestor de paquets es compon de l'executable 'pear' al qual podem proporcionar un conjunt de paràmetres segons les accions que vulguem realitzar:

```
$ pear list
Installed packages:
===================
Package         Version   State
Archive_Tar     1.2       stable
Console_Getopt  1.2       stable
DB              1.6.8     stable
http            1.3.3     stable
Mail            1.1.4     stable
Net_SMTP        1.2.6     stable
Net_Socket      1.0.5     stable
PEAR            1.3.4     stable
PhpDocumentor   1.3.0RC3  beta
XML_Beautifier  1.1       stable
XML_Parser      1.2.2     stable
XML_RPC         1.1.0     stable
XML_Util        1.1.1     stable
```

PEAR (i PHP) ja ve amb un conjunt de paquets instal·lats, cosa que es denomina el PFC (PHP *foundation classes*). Aquests paquets proporcionen a PHP la funcionalitat mínima necessària perquè PEAR funcioni i perquè disposem de les llibreries bàsiques de PHP.

A continuació presentem les opcions més habituals de PEAR:

Ordre	Resultat
`pear list`	Llista dels paquets instal·lats.
`pear list-all`	Llista de tots els paquets disponibles en PEAR.
`pear list-upgrades`	Llista dels paquets instal·lats amb actualització disponible.
`pear info <paquet>`	Proporciona informació sobre el paquet.
`pear install <paquet>`	Baixa i instal·la el paquet.
`pear search <text>`	Busca paquets en el dipòsit PEAR.
`pear upgrade <paquet>`	Actualitza el paquet si és necessari.
`pear upgrade-all`	Actualitza tots els paquets instal·lats amb actualització disponible.
`pear uninstall <paquet>`	Desinstal·la el paquet.

1.4.1. Capa d'abstracció del motor de la base de dades

Sembla evident que, per a la immensa majoria d'aplicacions basades en PHP, l'ús de la seva llibreria nativa d'accés a bases de dades condicionarà l'SGBD a usar amb l'aplicació. En aplicacions comercials, o que no poden ni volen estar tancades a un únic motor, no serà imprescindible disposar d'unes funcions que encapsulin la comunicació amb l'SGBD i que siguin independents d'aquest en les interfícies que ofereixen, mentre que internament cridaran les funcions natives de l'SGBD concret amb què s'estigui treballant en cada moment.

Així doncs, i buscant en PEAR, trobem el mòdul 'DB' una capa d'abstracció i encapsulament de la comunicació amb l'SGBD. En haver d'incorporar totes les funcionalitats dels motors que suporta, el resultat serà sempre el mínim conjunt de prestacions comunes a tots els SGBD. Les prestacions més destacades que ofereix la versió actual 1.6.8 són les següents:

Versions

La versió 1.6.8. era la més actualitzada en el moment d'elaboració d'aquest material (final del 2004).

- Interfície orientada a objectes.
- Una sintaxi comuna per a identificar SGBD i cadenes de connexió.
- Emulació de "sentències preparades" en els motors que no les suporten.
- Codis d'errors comuns.
- Emulació de seqüències o autoincrements en SGBD que no els suporten.
- Suport per a transaccions.
- Interfície per a obtenir informació de la metadada (informació sobre la taula o la base de dades).
- Compatibilitat amb PHP4 i PHP5.
- Motors suportats: dbase, fbsql, interbase, informix, msql, mssql, mysql, mysqli, oci8, odbc, pgsql, sqlite i sybase.

```
1 <?php
2 // Incloem la llibreria una vegada instal·lada mitjançant PEAR
3 require_once 'DB.php';
4
5 // Creem la connexió a la base de dades, en aquest cas PostgreSQL
6 $db =& DB::connect('pgsql://usuari:password@servidor/basededades');
7
8 // Comprovem error en la connexió
9 if (DB::isError($db)) {
10  die($db->getMessage());
11 }
12
13 // Realitzem la consulta:
14 $res =& $db->query('SELECT * FROM clients');
15
16 // Comprovem que la consulta s'ha realitzat correctament
```

```
17 if (DB::isError($res)) {
18 die($res->getMessage());
19 }
20
21 // Iterem sobre els resultats
22 while ($row =& $res->fetchRow()) {
23   echo $row[0] . "\n";
24 }
25
26 // Alliberem el full de resultats
27 $res->free()
28
29 // Desconnectem de la base de dades
30 $db->disconnect();
31 ?>
```

L'estructura del codi, i fins i tot la sintaxi de les sentències, és similar als exemples natius vistos anteriorment, exceptuant les parts de les sentències que feien referència al motor de base de dades en particular.

A continuació, avançarem pel codi ampliant la informació sobre cada pas.

La connexió s'especifica mitjançant una sintaxi de tipus DSN (*data source name*). Els DSN admeten multitud de variants, depenent del motor al qual ens connectem, però en gairebé tots els casos, tenen la forma següent:

```
motorphp://usuari:contrasenya@servidor/basededades?opcio=valor
```

- Connexió a MySQL

  `mysql://usuari:password@servidor/basededades`

- Connexió a MySQL per mitjà d'un *socket* UNIX:

  `mysql://usuari:password@unix(/cami/al/socket)/basededades`

- Connexió a PostgreSQL

  `pgsql://usuari:password@servidor/basededades`

- Connexió a PostgreSQL en un port específic:

  `pgsql://usuari:password@tcp(servidor:1234)/basededades`

En qualsevol crida a un mètode del paquet DB, aquest pot tornar l'objecte que li correspon (un full de resultats, un objecte representant la connexió, etc.) o bé un objecte que representi l'error que ha tingut la crida. D'aquesta manera, per a comprovar els errors que pot originar cada sentència o intent de connexió, n'hi haurà prou de comprovar el tipus de l'objecte tornat:

```
8  // Comprovem error en la connexió
9  if (DB::isError($db)) {
10   die($db->getMessage());
11 }
```

La classe DB_Error ofereix diversos mètodes. Malgrat que el més utilitzat és getMessage(), getDebugInfo() o getCode() poden ampliar la informació sobre l'error.

Per a realitzar consultes, disposem de dos mecanismes diferents:

Enviar la consulta directament a l'SGBD.

Indicar el gestor que prepari l'execució d'una sentència SQL i posteriorment indicar-li que l'executi una o més vegades.

En la majoria dels casos, optarem pel primer mecanisme i podrem procedir com segueix:

```
13 // Realitzem la consulta:
14 $res =& $db->query('SELECT * FROM clients');
```

Exemple

Pensem en un conjunt d'actualitzacions o insercions seguides o en un conjunt de consultes en què anem canviant un interval de dates o l'identificador del client sobre el qual es realitzen.

De vegades ens podem trobar executant la mateixa consulta diverses vegades, amb canvis en les dades o en el valor de les condicions. En aquests casos, és més indicat utilitzar el segon mecanisme.

Suposem que hem d'inserir un conjunt de clients en la nostra base de dades. Les sentències que executaríem serien semblants a les següents:

```
INSERT INTO Clients (nom, nif) VALUES ('José Antonio Ramírez','29078922Z');
INSERT INTO Clients (nom, nif) VALUES ('Míriam Rodríguez','45725248T');
...
```

En lloc d'això, podem indicar al motor de la base de dades que prepari la sentència, de la manera següent:

```
$sth = $db->prepare('INSERT INTO Clients (nom,nif) VALUES (?,?)');
```

Utilitzarem la variable $sth que ens ha tornat la sentència prepare cada vegada que executem el *query*:

```
$db->execute($sth, 'José Antonio Ramírez','29078922Z');
$db->execute($sth, 'Míriam Rodríguez','45725248T');
```

També podem passar un *array* amb tots els valors:

```
$datos = array('Míriam Rodríguez','45725248T');
$db->execute($sth, $datos);
```

I tenim l'opció de passar un *array* de dues dimensions amb totes les sentències que s'han d'executar, mitjançant el mètode `executeMultiple()`:

```
$totesDades = array(array('José Antonio Ramírez','29078922Z'),
        array('Míriam Rodríguez','45725248T'));
$sth = $db->prepare('INSERT INTO Clients (nom,nif) VALUES (?, ?)');
$db->executeMultiple($sth, $totesDades);
```

A continuació, examinarem els mètodes d'iterar sobre els resultats. En l'exemple inicial hem utilitzat la funció `fetchRow()` de la manera següent:

```
21 // Iterem sobre els resultats
22 while ($row =& $res->fetchRow()) {
23   echo $row[0] . "\n";
24 }
```

Però també disposem de la funció `fetchInto()`, que rep com a paràmetre l'*array* on volem que s'emmagatzemi el resultat:

```
21 // Iterem sobre els resultats
22 while ($res->fetchRow()) {
23   echo $row[0] . "\n";
24 }
```

Tant `fetchRow()` com `fetchInto()` accepten un altre paràmetre per indicar el tipus d'estructura de dades que emmagatzemarà en `$row`:

- DB_FETCHMODE_ORDERED: és l'opció per defecte. Emmagatzema el resultat en un *array* amb índex numèric.

- DB_FETCHMODE_ASSOC: emmagatzema el resultat en un *array* associatiu en què les claus són el nom del camp.

- DB_FETCHMODE_OBJECT: emmagatzema el resultat en un objecte on disposarem d'atributs amb el nom de cada camp per a obtenir el valor en cada iteració.

```
21 // Iterem sobre els resultats en mode associatiu
22 while ($res->fetchInto (($row,DB_FETCHMODE_ASSOC)) {
23    echo $row['nom'] . "\n";
24    echo $row['nif'] . "\n";
25 }
```

o bé:

```
21 // Iterem sobre els resultats en mode objecte
22 while ($res->fetchInto (($row,DB_FETCHMODE_OBJECT)) {
23    echo $row->nom . "\n";
24    echo $row->nif . "\n";
25 }
```

La funció `fetchInto()` accepta un tercer paràmetre per a indicar el número de fila que volem obtenir, en cas que no vulguem iterar sobre la mateixa funció:

```
21 // Obtenim la tercera fila de la consulta
22 $res->fetchInto ($row,DB_FETCHMODE_ASSOC,3); {
23    echo $row->nom . "\n";
24    echo $row->nif . "\n";
```

Hi ha altres mètodes per a obtenir diferents vistes del resultat com els següents:

- `getAll()`: obté un *array* de dues dimensions amb tot el full de resultats. Tindria una forma com la següent:

```
Array
(
   [0] => Array
      (
         [cf] => Joan
         [nf] => 5
         [df] => 1991-01-11 21:31:41
      )
   [1] => Array
   (
      [cf] => Kyu
      [nf] => 10
      [df] => 1992-02-12 22:32:42
   )
)
```

- `getRow()`: torna només la primera fila del full de resultats.
- `getCol()`: torna només la columna indicada del full de resultats.

De la mateixa manera que en les llibreries natives, hi ha mètodes que proporcionen informació sobre la consulta per si mateixa:

- `numRows()`: nombre de files del full de resultats.
- `numCols()`: nombre de columnes del full de resultats.
- `affectedRows()`: nombre de files de la taula afectades per la sentència d'actualització, inserció o esborrament.

1.4.2. Transaccions

PEAR::DB proporciona mecanismes per a tractar les transaccions independentment de l'SGBD amb què treballem.

Ja hem vist que l'operativa amb les transaccions està relacionada amb les sentències `begin`, `commit` i `rollback` de SQL. PEAR::DB inclou aquestes sentències en mètodes propis de la manera següent:

```
// Desactivem el comportament de COMMIT automàtic.
$db->autocommit(false);
..
..
if (...) {
   $db->commit();
} else {
   $db->rollback();
}
```

Atenció

En MySQL només funcionarà el suport de transaccions si la base de dades està emmagatzemada amb el mecanisme InnoDB.

En PostgreSQL no hi ha cap restricció.

Enlloc...

... no es fa un `begin`. En desactivar `l'autocommit` (que està activat per defecte) totes les sentències passaran a formar part d'una transacció, que es registrarà com a definitiva en la base de dades en cridar el mètode `commit()` o bé es rebutjarà en cridar el mètode `rollback()`, tornant la base de dades a l'estat en què estava després de l'últim `commit()`.

1.4.3. Seqüències

PEAR::DB incorpora un mecanisme propi de seqüències (AUTO_INCREMENT en MySQL), que és independent de la base de dades utilitzada i que pot ser de gran utilitat en identificadors, claus primàries, etc. L'únic requisit és que s'usin els seus mètodes de treball amb seqüències, sempre que s'estigui treballant amb aquesta base de dades; és a dir, no s'ha de crear la seqüència en l'SGBD i, després, treballar amb aquesta amb els mètodes que ofereix PEAR::DB. Si la seqüència la creem mitjançant la base de dades, llavors haurem de treballar amb ella amb les funcions esteses d'SQL que ens proporcioni aquest SGBD (en Post-

greSQL la funció `nextval()` o en MySQL la inserció del valor 0 en un camp AUTO_INCREMENT).

Disposem de les funcions següents per a treballar amb seqüències:

- `createSequence($nom_de_sequencia)`: crea la seqüència o torna un objecte DB_Error en cas contrari.
- `nextId($nom_de_sequencia)`: torna el següent identificador de la seqüència.
- `dropSequence($nom_de_sequencia)`: esborra la seqüència.

```
// Creem la seqüència:
$tmp = $db->createSequence('mySequence');
if (DB::isError($tmp)) {
  die($tmp->getMessage());
}

// Obtenim el següent identificador
$id = $db->nextId('mySequence');
if (DB::isError($id)) {
  die($id->getMessage());
}

// Usem l'identificador en una sentència
$res =& $db->query("INSERT INTO lamevaTaula (id, text) VALUES ($id, 'Hola')");

// Esborrem la seqüència
$tmp = $db->dropSequence('mySequence');

if (DB::isError($tmp)) {
  die($tmp->getMessage());
}
```

Finalment, en l'aspecte relacionat amb les metadades de les taules, PEAR::DB ofereix la funció `tableInfo()`, que proporciona informació detallada sobre una taula o sobre les columnes d'un full de resultats obtingut d'una consulta:

```
$info = $db->tableInfo('nomtaula');
print_r($info);
```

O bé:

```
$res =& $db->query('SELECT * FROM nomtaula');
$info = $db->tableInfo($res);
print_r($info);
```

El resultat serà similar al següent:

```
[0] => Array (
  [table] => nomtaula
  [name] => nom
```

```
    [type] => string
    [len] => 255
    [flags] =>
  )
  [1] => Array (
    [table] => nomtaula
    [name] => nif
    [type] => string
    [len] => 20
    [flags] => primary key not null
  )
```

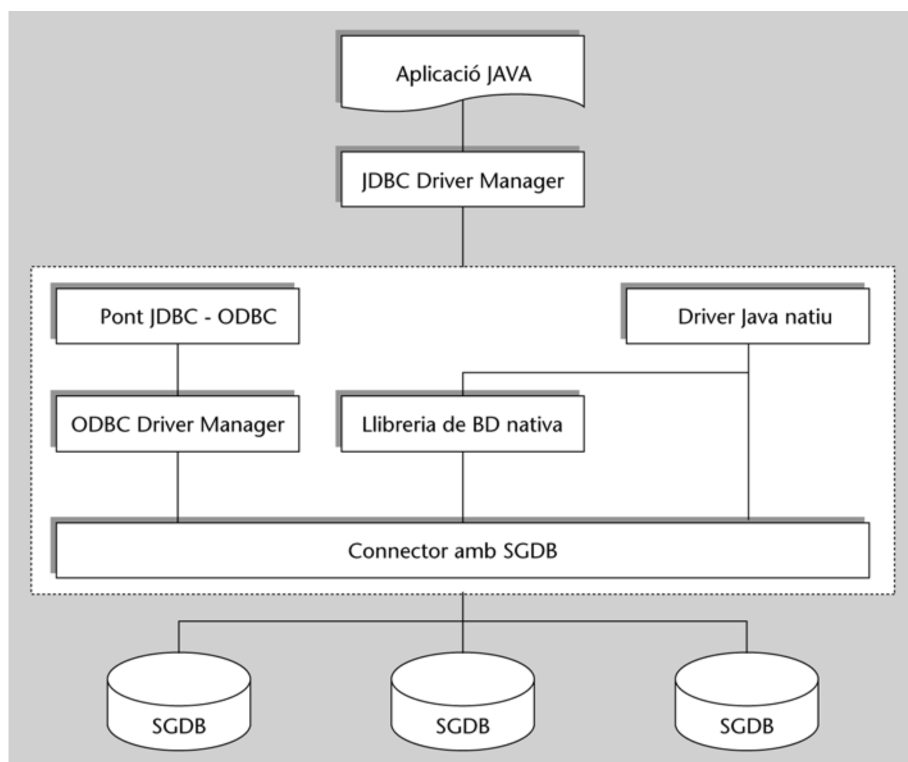
Funcionalitats

Hi ha funcionalitats més avançades d'aquesta llibreria que augmenten contínuament. De tota manera, amb les presentades n'hi ha prou per a identificar els avantatges de treballar amb una capa d'abstracció del motor de base de dades on s'emmagatzemen les dades de la nostra aplicació.

2. Connexió i ús de bases de dades en llenguatge Java

L'accés a bases de dades des de Java es realitza mitjançant l'estàndard JDBC (*Java data base conectivity*), que permet un accés uniforme a les bases de dades independentment de l'SGBD. D'aquesta manera, les aplicacions escrites en Java no necessiten conèixer les especificacions d'un SGBD en particular, n'hi ha prou de comprendre el funcionament de JDBC. Cada SGBD que es vulgui utilitzar amb JDBC ha de tenir un adaptador o controlador.

L'estructura de JDBC es pot expressar gràficament com segueix:



Hi ha *drivers* per a la majoria d'SGBD, tant de programari lliure com de codi obert. A més, hi ha *drivers* per a treballar amb altres tipus de dades (fulls de càlcul, fitxers de text, etc.) com si fossin SGBD sobre els quals podem realitzar consultes SQL.

Per a usar l'API JDBC amb un SGBD en particular, necessitarem el *driver* concret del motor de base de dades, que hi ha entre la tecnologia JDBC i la base de dades. Depenent de múltiples factors, el *driver* pot estar escrit completament en Java, o bé haver usat mètodes JNI (*Java native interface*) per a interactuar amb altres llenguatges o sistemes.

L'última versió de desenvolupament de l'API JDBC proporciona també un pont per a connectar-se a SGBD que disposin de *drivers* ODBC (*open database*

connectivity). Aquest estàndard és molt comú sobretot en entorns Microsoft i només s'hauria d'usar si no disposem del *driver* natiu per al nostre SGBD.

En el cas concret de MySQL i PostgreSQL, no tindrem cap problema a trobar els drivers JDBC:

- MySQL Connector/J: és el *driver* oficial per a MySQL i es distribueix sota llicència GPL. És un *driver* natiu escrit completament en Java.

- JDBC per a PostgreSQL: és el *driver* oficial per a PostgreSQL i es distribueix sota llicència BSD. És un *driver* natiu escrit completament en Java.

Tant l'un com l'altre, en la seva distribució en format binari, consisteixen en un fitxer .jar (*Java arxive*) que hem de situar en el CLASSPATH del nostre programa per a poder incloure'n les classes.

Java inclou la possibilitat de carregar classes de manera dinàmica. Aquest és el cas dels controladors de bases de dades: abans de realitzar qualsevol interacció amb les classes de JDBC, cal registrar el controlador. Aquesta tasca es realitza amb el codi següent:

```
String controlador = “com.mysql.jdbc.Driver”
Class.forName(controlador).newInstance();
```

o bé:

```
Class.forName(“org.postgresql.Driver”);
```

A partir d'aquest moment, JDBC està capacitat per a interactuar amb MySQL o PostgreSQL.

2.1. Accedir a l'SGBD amb JDBC

La interfície JDBC està definida en la llibreria java.sql. Importarem a la nostra aplicació Java totes les classes definides en aquesta:

```
import java.sql.*;
```

Atès que JDBC pot realitzar connexions amb múltiples SGDB, la classe `DriverManager` configura els detalls de la interacció amb cada un en particular. Aquesta classe és la responsable de realitzar la connexió, lliurant un objecte de la classe `Connection`.

```
String url="jdbc:mysql://localhost/demo";
String usuari="jo"
String contasenya="contrasenya"
Connection connexio = DriverManager.getConnection (url,usuari,contrasenya);
```

La destinació de la connexió s'especifica mitjançant un URL de JDBC amb la sintaxi següent:

```
jdbc:<protocol_sgbd>:<subnom>
```

La part `protocol_sgdb` de l'URL especifica el tipus d'SGBD amb el qual es realitzarà la connexió, la classe `DriverManager` carregarà el mòdul corresponent a aquest efecte.

El `subnom` té una sintaxi específica per a cada SGDB que tant per a MySQL com per a PostgreSQL és *//servidor/base_de_dades*.

Les sentències en JDBC també són objectes que haurem de crear a partir d'una connexió:

```
Statement sentenciaSQL = connexio.createStatement();
```

En executar una sentència, l'SGBD lliura uns resultats que JDBC també representa en forma d'objecte, en aquest cas de la classe `ResultSet`:

```
ResultSet res = sentenciaSQL.executeQuery("SELECT * FROM taula");
```

La variable *res* conté el resultat de l'execució de la sentència, i proporciona un cursor que permet llegir les files una a una.

Per a accedir a les dades de cada columna del full de resultats, la classe `ResultSet` disposa de diversos mètodes segons el tipus de la informació de la columna:

getArray()	getInt()
getClob()	getBoolean()
getString()	getLong()
getAsciiStream()	getByte()
getDate()	getObject()
getTime()	getObject()
getBigDecimal()	getBytes()
getDouble()	getBytes()
getTimestamp()	getRef()
getBinaryStream()	getRef()
getFloat()	getCraracterStream()
getURL()	getShort()
getBlob()	

```
import java.sql.*;
// Atenció, no s'ha d'importar com.mysql.jdbc, ja que es carrega dinàmicament!!

public static void main(String[] args) {

  // Carreguem el driver JDBC per a MySQL
  String controlador = "com.mysql.jdbc.Driver"
  Class.forName(controlador).newInstance();

  // Connectem amb la BD
  String url="jdbc:mysql://localhost/uoc";
  String usuari="jo"
  String contrasenya="contrasenya"
  Connection connexio = DriverManager.getConnection (url,usuari,contrasenya);

  // Creem una sentència SQL
  Statement sentenciaSQL = connexio.createStatement();
  // Executem la sentència
  ResultSet res = sentenciaSQL.executeQuery("SELECT * FROM taula");

  // Iterem sobre el full de resultats
  while (res.next()) {
    // Obtenim el camp 'nom' en forma de String
    System.out.println(res.getString("nom") );
  }

  // Finalment, s'alliberen els recursos utilitzats.
  res.close();
  sentencia.close();
  connexio.close();
}
```

En l'exemple anterior no s'ha previst res per a tractar els errors que es puguin produir, perquè en Java el tractament d'errors es fa per mitjà d'`Exceptions`. En JDBC s'han previst excepcions per als errors que es poden produir al llarg de tot l'ús de l'API: connexió, execució de la sentència, etc.

Revisem l'exemple, utilitzant excepcions per a tractar els errors.

```
import java.sql.*;
// Atenció, no s'ha d'importar com.mysql.jdbc, ja que es carrega dinàmicament!!

public static void main(String[] args) {

  try {
    // Carreguem el driver JDBC per a MySQL
    String controlador = "com.mysql.jdbc.Driver"
    Class.forName(controlador).newInstance();
  } catch (Exception e) {
    System.err.println("No puc carregar el controlador de MySQL ...");
    e.printStackTrace();
  }

  try {
    // Connectem amb la BD
    String url="jdbc:mysql://localhost/uoc";
    String usuari="jo"
    String contrasenya="contrasenya"
    Connection connexio = DriverManager.getConnection (url,usuari,contrasenya);
```

```
        // Creem una sentència SQL
        Statement sentenciaSQL = connexio.createStatement();
        // Executem la sentència
        ResultSet res = sentenciaSQL.executeQuery("SELECT * FROM taula");

        // Iterem sobre el full de resultats
        while (res.next()) {
          // Obtenim el camp 'nom' en forma de String
          System.out.println(res.getString("nom") );
        }

        // Finalment, s'alliberen els recursos utilitzats.
        res.close();
        sentencia.close();
        connexioSQL.close();
      } catch (SQLException e) {
        System.out.println("Excepció del SQL: " + e.getMessage());
        System.out.println("Estat del SQL: " + e.getSQLState());
        System.out.println("Error del Proveïdor: " + e.getErrorCode());
      }
    }
```

Mentre que l'operació `executeQuery()` de la classe `Statement` torna un objecte `ResultSet`, l'operació `executeUpdate`() només torna el seu èxit o fracàs. Les sentències SQL que s'utilitzen amb `executeUpdate`() són **insert**, **update**, o **delete**, perquè no tornen cap resultat.

Errors

Tots els errors de JDBC s'informen per mitjà d'`SQLException`.

`SQLWarning` presenta les advertències d'accés a les bases de dades.

```
public void Insertar_persona(String nom, adreca, telefon){
Statement sentencia = connexio.CreateStatement();
sentencia.executeUpdate( "insert into persones values("
+ nom + ","
+ cognom + ","
+ telefon + ")" );
}
```

2.2. Sentències preparades

Les sentències preparades de JDBC permeten la "precompilació" del codi SQL abans de ser executat, permetent consultes o actualitzacions més eficients. En el moment de compilar la sentència SQL, s'analitza quina és l'estratègia adequada segons les taules, les columnes, els índexs i les condicions de recerca implicats. Aquest procés, òbviament, consumeix temps de processador, però en realitzar la compilació una sola vegada, s'aconsegueix millorar el rendiment en següents consultes iguals amb valors diferents.

Un altre avantatge de les sentències preparades és que permeten la parametrització: la sentència SQL s'escriu una vegada, indicant-hi les posicions de les dades que canviaran i, cada vegada que s'utilitzi, li proporcionarem els arguments necessaris que seran substituïts als llocs corresponents. Els paràmetres s'especifiquen amb el caràcter '? '.

```
public class Actualitzacio{
  private PreparedStatement sentencia;

  public void prepararInsercio(){
    String sql = "insert into persones values ( ?, ? ,? )";
    sentencia = connexio.prepareStatement(sql);
  }

  public void insertarPersona(String nom, adreca, telefon) {
    sentencia.setString(1, nom);
    sentencia.setString(2, adreça);
    sentencia.setString(3, telefon);
    sentencia.executeUpdate();
  }
}
```

En utilitzar aquesta classe, òbviament, haurem de cridar primer el mètode que prepara la inserció i, posteriorment, cridar tantes vegades com sigui necessari el mètode `inserirPersona`.

Es defineixen tres paràmetres en la sentència SQL, als quals es fa referència mitjançant nombres enters consecutius:

```
String sql = "insert into persones values ( ?, ? ,? )";
```

Exemple

El segon dels tres paràmetres s'especifica de la manera següent
`sentencia.setString(2, adreca);`

La classe `PreparedStatement` inclou un conjunt d'operacions de la forma `setXXXX()`, on `XXXX` és el tipus de dada per als camps de la taula. Una d'aquestes operacions és precisament `setString()` que insereix la variable en un camp de tipus cadena.

2.3. Transaccions

L'API JDBC inclou suport per a transaccions, de manera que es pugui desfer un conjunt d'operacions relacionades en cas necessari. Aquest comportament és responsabilitat de la classe `Connection`.

Per omissió, cada sentència s'executa en el moment en què se sol·licita i no es pot desfer. Podem canviar aquest comportament amb l'operació següent:

```
conexion.setAutoCommit(false);
```

Després d'aquesta operació, és necessari cridar `commit()` perquè totes les sentències SQL pendents es facin definitives:

```
sentencia.executeUpdate(...);
...
sentencia.executeUpdate(...);
...
connexio.commit(); // Les dues actualitzacions anteriors es fan permanents
```

En cas contrari, desfarem totes les actualitzacions després de l'últim `commit()`:

```
sentencia.executeUpdate(...);
...
sentencia.executeUpdate(...);
...
sentencia.executeUpdate(...);
...
connexio.rollback(); // Cancel·la les tres últimes actualitzacions
```

Resum

Hem presentat algunes de les maneres més habituals de connectar-se als SGBD que hem vist en mòduls anteriors des de PHP i Java.

Hem pogut comprovar que no hi ha gaires variacions ni restriccions entre MySQL i PostgreSQL quant al seu accés des de llenguatges de programació, sinó al contrari, els esforços s'encaminen a homogeneïtzar el desenvolupament i independitzar-lo de l'SGBD amb què treballem.

En PHP, hem repassat els mètodes natius i vist les seves particularitats. Hem comprovat que, tret que necessitem característiques pròpies i molt avançades d'un SGBD, no és aconsellable usar aquests mètodes pels problemes que ens pot ocasionar un canvi de gestor de base de dades en el futur. Tot i així, és interessant revisar-los perquè trobarem moltes aplicacions de programari lliure desenvolupades en PHP que els utilitzen.

PEAR::DB és un exemple de llibreria d'abstracció (no és l'únic) ben feta i amb el suport de la fundació que manté PHP. Té tot el que podem desitjar i actualment és completament estable i usable en entorns empresarials.

En Java, hem vist JDBC. Encara que l'API dóna molt més de si, creiem que hem complert els objectius d'aquest apartat, sense entrar en conceptes que només programadors experts en Java podrien apreciar.

Així doncs, s'han proporcionat els elements de referència i els exemples necessaris per a treballar amb bases de dades en les nostres aplicacions.

Bibliografia

DBC Technology: http://java.sun.com/products/jdbc/

Documentació de PHP: http://www.php.net/docs.php

MySQL Connector/J: http://dev.mysql.com/downloads/connector/j

PEAR::DB Database Abstraction Layer: http://pear.php.net/package/DB

PEAR :: The PHP Extension and Application Repository: http://pear.php.net/

PostgreSQL JDBC Driver: http://jdbc.postgresql.org/