A Crash Course in Perl5

Part 8: Database access in Perl

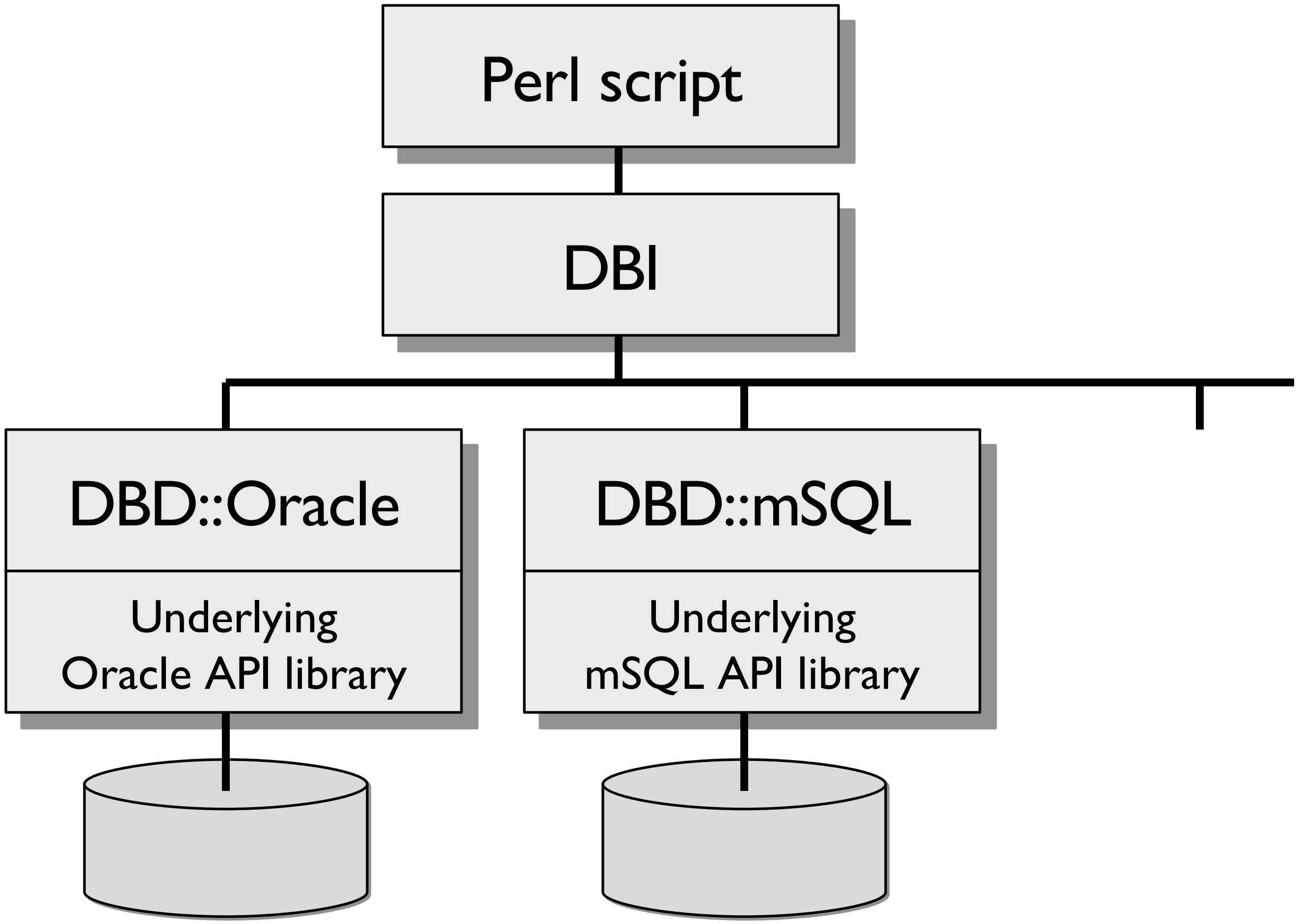
Zeegee Software Inc.
http://www.zeegee.com/

Terms and Conditions

What is Perl DBI?

- Perl's "DBI" is an API that allows users to access multiple database types transparently.
- E.g., if you connecting to an Oracle, Informix, mSQL, Sybase or whatever database, you **mostly** don't need to know the underlying mechanics of the 3GL layer. The API defined by DBI will work on all these database types.
- You can even connect to two different databases of different vendor within the one perl script, e.g., for data migration. *Even CSV files are supported!*

How it works


Perl script
DBI
DBD::Oracle
Underlying Oracle API library
DBD::mSQL
Underlying mSQL API library

What drivers do I have?

- You can get a list of all the available drivers installed on your machine by using the **available_drivers** method:

```
#!/usr/bin/perl -w
use strict;                           # always a good idea
use DBI;                              # load the Perl DBI modules

foreach (DBI->available_drivers) {
        print "$_\n";
}
```

- Returns a list with each element containing the "data source prefix" of an installed driver (e.g., "dbi:Oracle:"). More about these later...

Connecting to the database

- Connecting to different databases requires different techniques. For exhaustive information, be sure to read the documentation that comes with your DBD.
- This example will cover connecting to Oracle...

A sample connection

- To obtain a *database handle*, use the DBI->connect method like this:

```
use strict;
use DBI;

my $dbh = DBI->connect('dbi:Oracle:somedb',
                       'myusername',
                       'mypassword')
          || die "connect failed: $DBI::errstr\n";
```

- Let's examine this in detail...

Connecting

DBI->connect

- General form...

```
$dbh = DBI->connect( $data_source,
                     $username,
                     $password,
                     \%attr);
```

- If the *$data_source* is undefined or empty, DBI will use the environment variable **$DBI_DSN**.
- If *$username/$password* are undefined, defaults to environment values **$DBI_USER** and **$DBI_PASS**.

The data source name

- The **data source name** ("DSN") is like a URL.
  It takes the general form:

  dbi:***drivername***:***instance***

- The **driver name** in our example is "Oracle", because we're using DBD::Oracle to connect.
- The **instance** is the database instance we want to connect to. This part is driver-dependent...

Some data source names

- Some popular data source name formats:

```
dbi:driver:db
dbi:driver:db@host:port
dbi:driver:database=db;host=host;port=port
```

⚠ There is no standard for the text following the driver name. Each driver can use whatever syntax it wants! *Read the documentation for your driver* (DBD::Oracle)

- Last one is ODBC-style; generally preferred among authors, so try that if all else fails. :-)

Connecting

Data source names for Oracle

- With DBD::Oracle, the DBI->connect DSN can be one of the following:

```
dbi:Oracle:tnsname

dbi:Oracle:sidname

dbi:Oracle:host=hostname;sid=sid
```

- Some other less common formats also work if supported by the Oracle client version being used.
- DBD::Oracle supports an unlimited number of concurrent database connections to one or more databases.

Did the connect succeed?

- The connect method returns an database handle object (true) on success, and undef (false) otherwise.
- If it failed, we can check **$DBI::errstr** for the reason.

```
# Connect to the database, obtaining a handle.
# Make sure to verify that we succeeded!
my $dbh = DBI->connect('dbi:Oracle:somedb',
                       'myusername',
                       'mypassword')
          || die "connect failed: $DBI::errstr\n";
```

Options for DBI->connect

- DBI->connect takes a fourth argument, a ref to a hash of options:

```
$dbh = DBI->connect($datasource, $user, $pass, {
                              RaiseError => 1,
                              AutoCommit => 1,
                            });
```

Options for DBI->connect

- **AutoCommit** says whether or not to automatically commit database transactions. If your database doesn't support transactions, it *must* be set true, or a fatal error will occur.

- **RaiseError** and **PrintError** control how errors are handled when they occur:

  – If **RaiseError** is true, we... `croak $DBI::errstr`

  – If **PrintError** is true, we just... `warn $DBI::errstr`

⚠ You are *strongly* encouraged to use RaiseError!

Embedding options in DSNs

- You can embed the connection options inside a data source name, using Perlish hash syntax:

```
"dbi:Oracle(PrintError=>0,Taint=>1):mydb"
```

- Individual attributes embedded in this way take precedence over any conflicting values given in the \%attr parameter. *In other words, the DSN wins.*

Disconnecting

- If you don't disconnect your handle explicitly, you'll get an error from the destructor: "*Database handle destroyed without explicit disconnect*".
- So remember to disconnect when you're done:

```
# Disconnect the handle from the database:
$dbh->disconnect;
```

Error handling

- There are several different kinds of *handles* you may be manipulating:
  - **Database** handles ($dbh), returned by DBI->connect
  - **Statement** handles ($sth), returned by $dbh->prepare
  - **Driver** handles ($drh) (rarely seen)
- For the purposes of error handling, these are all treated identically: there's a nice, consistent interface for getting the last error that happened on a given handle...

Error handling

$h->err

- **$h->err** returns the **error number** that is associated with the current error flagged against the handle $h.
- Usually an integer, but don't depend on that!
- The error number depends completely on the underlying database system: switch from Oracle to MySQL, and the numbers will be different! *Think about portability.*
- *Example*: an Oracle connection failure of ORA-12154 may cause $h->err to return 12154.

Error handling

$h->errstr

- **$h->errstr** returns a textual description of the error, as provided by the underlying database.
- Corresponds to the number returned by $h->err.
- *Example:* the Oracle error above returns something like...

"ORA-12154: TNS:could not resolve service name"

Error handling

$h->state

- **$h->state** returns a string in the format of the standard SQLSTATE five-character error string.
- The success code "**00000"** is translated to 0 (false) as a special case.
- *Many drivers do not fully support this method.*
  If unsupported, then state will return **"S1000"** (General Error) for all errors.
- Again, read the documentation for your DBD!

Error handling

Tracing

- To assist you in tracking down bugs, you can put a trace on DBI activity via **DBI->trace(*level*).** There are several valid tracing levels:

0	Disables tracing.
1	Traces DBI method calls, showing returned values & errors
2	As for 1, but also includes method entry with parameters.
3	As for 2, but also includes more internal driver information.
4	Levels 4 and above can include more detail than is helpful.

- **DBI->trace** takes an optional second argument: a file to which the trace information is appended.

Sending SQL statements

- Note that there are two types of SQL statement:
  - Statements which returns rows, like **select.**
    For these, we use the *prepare()* and *execute()* methods.
  - Statements which merely perform an action, like **create**.
    For these, we can just use the simple *do()* method.

Queries/commands

$dbh->do

```
# Create a string containing our SQL...
my $sql = <<EOF;
   CREATE TABLE employees (
      id INTEGER NOT NULL,
      name VARCHAR(64),
      phone CHAR(10)
   )
EOF

# ...and execute it:
$dbh->do($sql);
```

Queries/commands

Preparing a SELECT

- Just as we get a *database handle* when we connect to the database, we get a *statement handle* when we prepare a SQL statement for execution:

```
my $sql = "SELECT * FROM employees";
my $sth = $dbh->prepare($sql);
$sth->execute;
...
```

- This statement handle is what we work with to get back rows.

Reading the rows

- Once we do $sth->execute, we have many choices for how we want to get the rows back!

$sth->fetchrow_array	Get next row as ($col1, $col2, ...)
$sth->fetchrow_arrayref	Get next row as [$col1, $col2, ...]
$sth->fetchrow_hashref	Get next row as {'colname'=>$col1, ...}
$sth->fetchall_arrayref	Get *all* rows, each as arrayref or hashref
$sth->bind_col	Load next row directly into Perl variables

Queries/commands

$sth->fetchrow_array

- Returns the columns of the next row, and empty when done:

```
my $sth = $dbh->prepare(qq{SELECT id, name, phone
                           FROM employees});
$sth->execute();

my @row;
while (@row = $sth->fetchrow_array()) {
    my ($id, $name, $phone) = @row;
    ...
}
$sth->finish();
```

Queries/commands

$sth->fetchrow_arrayref

- Returns columns of the next row, undef when done.
- Returns same arrayref with different contents on each call: copy values elsewhere if keeping them!

```
my $sth = $dbh->prepare(qq{SELECT id, name, phone
                           FROM employees});
$sth->execute();

my $row;
while ($row = $sth->fetchrow_arrayref()) {
    my ($id, $name, $phone) = @$row;
    ...
}
$sth->finish();
```

Queries/commands

$sth->fetchrow_hashref

- Returns the columns of the next row in a {colname=>value} hash, and undefined when done:

```
my $sth = $dbh->prepare(qq{SELECT id, name, phone
                           FROM employees});
$sth->execute();

my $rowh;
while ($rowh = $sth->fetchrow_hashref()) {
    my $id    = $rowh->{'id'}
    my $name  = $rowh->{'name'}
    my $phone = $rowh->{'phone'};
    ...
}
$sth->finish();
```

Queries/commands

$sth->fetchall_arrayref

- To fetch just the first column of every row, as an arrayref

```
$all = $sth->fetchall_arrayref([0]);        # $all is ref to array of arrays
```

- To fetch the columns 0 and 2 of every row, as an arrayref:

```
$all = $sth->fetchall_arrayref([0,2]);      # $all is ref to array of arrays
```

- To fetch all fields of every row as a hash ref:

```
$all = $sth->fetchall_arrayref({});         # $all is ref to array of hashes
```

- To fetch only fields "id" and "name" of every row as a hash ref:

```
$all = $sth->fetchall_arrayref({'id'=>1, 'name'=>1 });
                                            # $all is ref to array of hashes
```

Queries/commands

$sth->bind_columns

- Another way to get back rows is by **binding** Perl variables to the columns of the results, then **fetch**ing rows one at a time until we're done:

```
my $sth = $dbh->prepare(qq{SELECT id, name, phone
                           FROM employees});
$sth->execute();

my $id, $name, $phone;
$sth->bind_columns(\$id, \$name, \$phone);
while ($sth->fetch()) {
   print "$id, $name, $phone\n";
}
$sth->finish();
```

Queries/commands

$sth->finish

- Indicates that no more data will be fetched from this statement handle before it is either executed again or destroyed.
- Rarely needed, but can be helpful in very specific situations to allow the server to free up resources (such as "sort" buffers).
- When all the data has been fetched from a SELECT statement, the driver should automatically call finish for you. So you should not normally need to call it explicitly.

$sth->rows

- Returns the number of rows affected by the last *row affecting* command, or -1 if the number of rows is not known or not available.
- You can only rely on a row count after a *non-SELECT* execute (for some specific operations like UPDATE and DELETE), or after fetching *all* the rows of a SELECT statement.
- Don't even depend on this giving you "rows-so-far" with SELECTs!

Prepared statements

- Parsing SQL is very time consuming!
- Best to *prepare* a statement with certain parts left "empty" (parsing it just once), and then substitute in the missing pieces when needed.
- We do this with the special **bind_param()** method...

Queries/commands

Prepared statements

```
use DBI qw(:sql_types);                 # for SQL_VARCHAR
...
my $sth = $dbh->prepare(qq{ SELECT id, name
                            FROM employees
                            WHERE name like ? });
foreach my $pattern ('Jean%', 'Joan%', 'June%') {
   $sth->bind_param(1, $pattern, SQL_VARCHAR);
   $sth->execute();

   $sth->bind_columns(undef, \$id, \$name);
   while ($sth->fetch) {
      print "$id, $name\n";
   }
}
```

Quoting strings

- To turn a Perl string into a SQL string appropriate for your database, use $dbh->quote:

```
my $unsafe = "Don't do it!";
my $safe = $dbh->quote($unsafe);

my $sth = $dbh->prepare(qq{
          SELECT *
          FROM msgs
          WHERE message = $safe
});
```

'Don''t do it!'

Transactions

- Suppose we have two tables, *employees* and *departments*, which have to be in synch: e.g., a new employee has an entry in both tables.
- We want to protect against situations where we corrupt our database by updating one table but then we quit for some reason before we update the other!
- DBMSs like Oracle support this through **transactions**... a transaction is a group of operations which must succeed collectively or else not at all.
- If the last operation succeeds, we **commit** the whole transaction; else, we **roll back** to the point before we started.

Using transactions

- If you plan to use transactions, then when you connect, be sure to ask that errors cause a thrown exception... and don't auto-commit!

```
my $dbh = DBI->connect('dbi:Oracle:somedb',
                       'myusername',
                       'mypassword',
                        {
                          RaiseError => 1,
                          AutoCommit => 0
                        });
```

- If/when RaiseError causes an exception to be thrown, we'll catch and handle it...

A skeleton for transactions

```
$dbh->{AutoCommit} = 0;  # enable transactions
$dbh->{RaiseError} = 1;  # make sure err raises exception
eval {
      foo(...)          # do lots of work here
      bar(...)          # including inserts
      baz(...)          # and updates
      $dbh->commit;    # commit changes if we make it here
};
if ($@) {
      warn "Transaction aborted because $@";
      $dbh->rollback; # undo the incomplete changes
      # add other application clean-up code here
}
```

Stored procedures

- DBD::Oracle can execute a block of PL/SQL code by starting it with BEGIN and ending it with END; we use PL/SQL blocks to call stored procedures.
- Here's a simple example that calls a stored procedure called ``foo'' and passes it two parameters:

```
$sth = $dbh->prepare("BEGIN foo(:1, :2) END;");
$sth->execute("Baz", 24);
```

A stored procedure call

- Here's a stored procedure called with two parameters and returning the return value of the procedure. The second parameter is defined as IN OUT, so we use **bind_param_inout** to enable it to update the Perl var:

```
$sth = $dbh->prepare(qq{BEGIN
                        :result = func_name(:id,:changeme)
                        END;});
$sth->bind_param(":id", "FooBar");
my ($result, $changeme) = (41, 42);
$sth->bind_param_inout(":result",  \$result,  100);
$sth->bind_param_inout(":changeme", \$changeme, 100);
$sth->execute();
print "returned '$result'; changed is '$changeme'\n";
```

Additional reading

- "Programming the Perl DBI" is the official book on the DBI written by Alligator Descartes and Tim Bunce. Published by O'Reilly & Associates, released on February 9th, 2000.
- The DBI:: manual page (try "perldoc DBI" on your system)