DBI - A Database Interface for perl5

Alligator Descartes
descarte@arcana.co.uk

This article was originally published in issue 5 of "The Perl Journal". It is republished courtesy of "The Perl Journal" in the form of the final unedited draft. Readers should consult the published version for absolute accuracy. The final published draft also included corrections made by Tim Bunce.

Abstract:

DBI, the Database Interface for perl5, is an ongoing effort to design and implement a database-independent interface for database connectivity which abstracts the low-level 'guts' of database technologies away from the programmer. With the explosion in popularity of perl as the de rigeur language for CGI programming, a simple, and standard, connection interface to databases is imperative.

The Architecture of DBI

The architecture of DBI is an elegant one. By the very concept of the interface that we are trying to define, we are channelled towards the solution. However, perl, as usual, helps us along the way by providing powerful syntactic constructs and regular expressions capabilities that are core to the usability and simplicity of data processing required by large database applications.

Figure 1: The Architecture of DBI

The DBI interface is the term used to describe both the interface specification, methods with which we write programs and the software modules that comprise the system. We shall firstly take a look at why you should be using DBI, then describe the...
structure of DBI and its interaction with perl. We finally present an example DBI.

Why DBI?

Database Independence

Database programming, as you will learn after you've programmed and/or a few different ones, is pretty much-of-a-muchness. Hence, DBI. The fundamental processes involved in doing anything with a database are pretty similar right across the board.

- Connection and Disconnection
- Opening and Closing cursors
- Fetching and Putting data

The typical order in which this happens is:

1. Load DBI driver
2. Connect to database which implicitly loads the required DBD
3. Open a cursor containing a SQL statement
4. Fetch the results
5. Close the cursor
6. Close the database connection
7. Exit

Now, between each database type, there are variances in details, like datatypes of fields retrieved, precision, extra non-ANSI-compliant features and so on, but the fundamentals are identical. DBI provides a unified layer at which people can write generally portable code, but, can access the non-standard features as well, if required.

Now, assuming we've fetched some data from the database, what next? Well, the good news is that DBI fetches data back into scalar variables, which are now subject to all the usual nice things perl can do with data.

Another extremely useful thing that can be achieved with DBI is to connect to a database simultaneously from within the same perl program, even, dare we say, to databases of separate vendor types, eg, you can connect to an Oracle and Informix database simultaneously from within one script. What's the point you cry. Well, say you have a corporate Oracle database that you want to publish some cut-down data from on the WWW. Now, Oracle via CGI can be slow due to Oracle's internal login procedures, so you've decided to use mSQL instead.

Option number 1 is to write a program that runs SQL scripts and dumps the say, comma-separated, which another program then reads and loads into the database. This is really stupid.

Option number 2 is to write a single perl script that connects simultaneously...
Oracle and the mSQL databases, reads the data from Oracle, and puts it in mSQL...transparently. Change Oracle to Informix and you won’t need to alter a thing.

Think on that the next time you're writing two separate programs in C with precompilers on separate platforms. Then think on the paradise of cross-platform and cross-database connectivity. You know it makes sense.

DBI is a current and living organism and, coupled with perl's current popula scripting language and rapid development tool, is set to become a more important decision to use perl as a “serious” programming language. Similarities have been observed between DBI and ODBC which have led to questions of “Why do we need DBI simpler than ODBC. DBI will run immediately on more platforms than ODBC this isn’t enough to convince you, DBI is currently being redesigned to allow ODBC-compliant code, which DBI will translate for you at run-time. ODBC will become another emulation layer supported by DBI!

**Databases vs. dbm**

Databases, be they Object DataBase Management Systems (ODBMS) or Relational DataBase Management Systems (RDBMS) are engineered to store and retrieve data. That's their prime purpose. UNIX was originally blessed with simple file databases, namely the dbm or DataBase Manager system. This system provided a capability to store data in files, and retrieve that data quickly. However, this system had several notable drawbacks:

**File Locking**

The dbm systems did not allow particularly robust file locking capability for correcting problems that may arise due to unsynchronized writes occurring.

**Arbitrary Data Structure**

Most importantly, the dbm systems only allowed a fixed data structure value. The value could be a complex object, eg a struct, but the key had to be unique.

This was a large limitation on the usefulness of dbm systems.

However, the dbm systems still provide a useful function for users with extremely simple datasets and limited resources, since they are fast and robust and extremely well-tested. Perl modules to access dbm systems have now been integrated into the core perl distribution, GDBM_File.

**The Modules**

perl5 has a powerful mechanism to 'plug' external modules into the perl interpreter. This mechanism is realised by actually compiling and linking the module into the interpreter or by Dynaloading, or dynamically loading, the module into a running interpreter as and when it is required. This system of plugging modules in is core to the philosophy of DBI.
DBI essentially acts as a conduit for the DBD modules. The DBDs all implement methods defined in DBI, eg `connect()`, but implement them in a database-specific way. To clarify this somewhat, since you, the DBI user, wishes to use a completely database-independent programming layer, some part of the system must execute the database-dependent code. This is what the DBD does. The DBD applications will never even know the DBD is there! All they will be aware of database-independent methods defined by DBI. The DBD code is written by many volunteers and now cover a fairly broad base of database vendors, in `Informix`, `mSQL`, `Ingres` and `Sybase`. The WWW pages listed in Section 6.2 detailed and up-to-date information on each module.

Before we even start looking at using DBI, you need to download and install the modules. You will always require the DBI module itself, as well as one of the DBD modules for whichever database you have installed. You can download DBI and the modules from the Comprehensive Perl Archive Network, or CPAN, details of which can be found in Section 6.2. Please follow the instructions in the files prefixed with README. They are called README for a good reason. Your ability to ask questions and get a civil answer depends on it!

The first step you need to take is to inform the perl interpreter that now is a good time to load the DBI module. This should always happen before any other DBI work, such as loading one of the database vendor-specific libraries, or attempting to connect to a database:

```perl
#!/usr/bin/perl -w
use DBI;
```

And that's it! Honest. If you don't believe me, try this:

```perl
$ PERL_DL_DEBUG=255 perl -e 'use DBI;'
```

See. That simple statement has Dynaloed the shared library containing the DBI module into the interpreter, and it has also imported the DBI interface methods, which means we can now start using DBI in anger.

If you refer back to the Architecture diagram in Figure 1, you can clearly see the databases are marshalled, or funnelled, through the DBI module. Therefore, we need to make perl aware of our DBDs. and to do this, we need to use DBI.

There are two ways to load drivers with DBI. One is the proper way to do it, the supported, clever and 'Don't annoy Tim Bunce' way of doing things is to use the `DBI->connect( )` method, specifying the driver that you wish to load as an additional argument. For example:

```perl
#!/usr/bin/perl -w
use DBI;
$dbh = DBI->connect( 'connection_string', 'username', 'password', 'mSQL' );
if ( !defined $dbh ) {
```

4 of 11 12/21/98 11:26 AM

which will load the DBI driver, then load the mSQL driver and attempt a database connection to the specified mSQL database. This call returns a database handle, which more of later. To use this method for other DBDs, simply change the 4 th argument from mSQL to whatever, eg, Oracle.

The second, sneaky and "Don't tell Tim" method is to use an internal method to DBI called install_driver(), which explicitly loads the driver and returns what is called a driver handle. example:

```perl
#!/usr/bin/perl -w
use DBI;
$drh = DBI->install_driver( 'mSQL' );
if ( !defined $drh ) {
    die "Cannot load driver: $!

This technique is quite useful for checking whether or not your version of perl has various drivers installed when you don't actually want to attempt a database connection. However, as I said, don't tell Tim.

Handles

Handles are perl objects returned by various DBI methods which the programmer can use to access data at various abstracted layers. The handles that are used by DBI and can be seen in Figure 2:

**Driver Handle**
A Driver Handle, or drh, points at a database type, eg, if you are connecting simply to an mSQL database, you should only ever have one driver handle. If you are connecting to an Oracle and an mSQL database, you will have two driver handles, as each driver handle does not connect you to a database, nor does it let you access data.
database operations. It merely acts as a conduit between the low-level calls and the DBI methods.

**Database Handle**

A Database Handle, or `dbh`, encapsulates a single connection to a given database handle. There can be any number of database handles per driver handle, if we have a script that copies data from one database to another databases are mSQL, then we will have 1 driver handle, and 2 database connect through the driver handle. In the case of having an mSQL database, we would have 2 driver handles, each with a single connection.

**Statement Handle**

A Statement Handle, or `sth`, encapsulates a statement issued to a database handle. As with database handles, there can be any number of statement handles per database handle. For example, if we have two tables in one containing data, the other containing a stale copy of the data, and program that refreshes the stale copy from the original, we could use handles, one to `SELECT` the data from the first table, the second to `UPDATE` the second. These statement handles would operate asynchronously, with the advent of multi-threading, this asynchronous behaviour would extremely powerful.

**Emulation Layers**

Some of the most commonly asked questions on the comp.lang.perl.* newsgroups and DBI mailing lists are ``I can't get oraperl to compile for perl5. It only seems to work with perl4. What do I do?'' and ``I've got all this Oraperl/Ingperl/isqlperl stuff and I want to upgrade to DBI, but I don't want to recode it all. What can I do?''

Well, as per usual, we're ahead of you. DBD::Oracle was released originally with an Oraperl emulation layer, i.e., a layer of software that translates the original Oraperl API calls into DBI method invocations. The basic upshot of this is that your existing Oraperl code will now work transparently using DBI and DBD::Oracle, which means you can now start writing new code using the DBI interface, whilst either maintaining the Oraperl code, or migrating it.

The DBD::mSQL driver is about to release an Msqlperl emulation layer, and has proposed isqlperl emulation layer in progress.

**Resources**

Several resources dedicated to DBI are available on the via the World Wide Web, anonymous ftp and mailing lists.

**DBI Mailing Lists**

The mailing lists related to DBI are run excellently by Ted Mellon. There are three lists to date, being `dbi-users`, for general chit-chat and support, `dbi-dev`, for developer ideas and `dbi-announce`, for announcements of new driver releases and so on. To join these lists, please see the URL:
and use the forms there. To join the mailing list via email, please see the *Frequently Asked Questions*.

**DBI WWW Pages**

The DBI WWW pages are located at:

http://www.arcana.co.uk/technologia/perl/DBI

and should be consulted at all opportunities. The *Frequently Asked Questions Specification* and pointers to documentation sources including the mailing list here, amongst other things.

**DBI ftp**

The DBI modules and drivers are available via anonymous ftp from:

ftp://ftp.demon.co.uk/pub/perl/db

The modules are also available via the WWW pages through CPAN, which may be reached via

http://www.perl.com/CPAN/

**Example Code**

We now present some simple example code using the DBI interface which should be consulted in conjunction with the documentation available at the resources.

**Basic Connection and Disconnection**

```
#!/usr/bin/perl -w
#
# (c)1996 Alligator Descartes <descarte@hermetica.com>
#
# inout.pl: Connects and disconnects from a specified database

use DBI;

if ( $#ARGV < 0 ) {
    die "Usage: inout.pl <Database String> <Database Vendor>\n";
}

# Create new database handle. If we can't connect, die()
$dbh = DBI->connect( '', $ARGV[0], '', $ARGV[1] );
if ( !defined $dbh ) {
    die "Cannot connect to mSQL server: $DBI::errstr\n";
}

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

**SELECTing data from the database**

#!/usr/bin/perl -w
#
# (c)1996 Alligator Descartes <descarte@hermetica.com>
#
# select.pl: Connects to a database called 'test' on a
given database, then SELECTs some basic data
# out in array and scalar forms

use DBI;

if ( $#ARGV < 0 ) {
    die "Usage: select.pl <Database String> <Database Vendor>\n"
}

# Create new database handle. If we can't connect, die()
$dbh = DBI->connect( '', $ARGV[0], '', $ARGV[1] );
if ( !defined $dbh ) {
    die "Cannot connect to mSQL server: $DBI::errstr\n";
}

# Prepare the statement for execution
$sth =
    $dbh->prepare( "
    SELECT id, name
    FROM table
    " );
if ( !defined $sth ) {
    die "Cannot prepare statement: $DBI::errstr\n";
}

# Execute the statement at the database level
$sth->execute;

# Fetch the rows back from the SELECT statement
while ( @row = $sth->fetchrow() ) {
    print "Row returned: @row\n";
}

# Re-execute the statement to bring the rows back again
$sth->execute;

# Fetch the data back into separate variables this time
while ( ( $id, $name ) = $sth->fetchrow() ) {
    print "ID: $id\tName: $name\n";
}

# Release the statement handle resources
$sth->finish;

# Disconnect from the database
$dbh->disconnect;
exit;

**Executing immediate statements**
#!/usr/bin/perl -w
#
# (c)1996 Alligator Descartes <descarte@hermetica.com>
#
# execute.pl: Connects to a database called 'test' on a
given database, then EXECUTEs an update
# statement. This is non-cursorial, so do() is used.

use DBI;

if ( $#ARGV < 0 ) {
    die "Usage: execute.pl <Database String> <Database Vendor>
";
}

# Create new database handle. If we can't connect, die()
$dbh = DBI->connect( '', $ARGV[0], '', $ARGV[1] );
if ( !defined $dbh ) {
    die "Cannot connect to mSQL server: $DBI::errstr";
}

# Prepare the statement for immediate execution
$rv =
    $dbh->do( "
        UPDATE table
        SET name = 'Alligator Descartes'
        WHERE id = 1
    ");
if ( !defined $rv ) {
    die "Statement execution failed: $DBI::errstr";
}

# Disconnect from the database
$dbh->disconnect;

exit;

About this document ...

DBI - A Database Interface for perl5

This document was generated using the LaTeX2HTML translator Version 96
University of Leeds.

The command line arguments were:
latex2html -split 0 dbi.

The translation was initiated by Alligator Descartes on Sun Jul 5 16:37:18 E

...cursors

A cursor is a database construct that is a statement that fetches multi-
back from a database, eg, a SELECT statement, ie, it is not an atomic c
Statements such as SELECT and INSERT are cursor-based, in that they
multiple rows, one at a time, whereas statements such as UPDATE and
atomic.
SQL
SQL (pronounced 'sequel') is an ANSI-compliant Standard Query Language for querying and manipulating data within databases.

required
Although this obviously makes the code less portable.

variables
Generally, but not necessarily the case. This is a simplification.

WWW

CGI
CGI, or the Common Gateway Interface allows WWW site designers to execute programs on the WWW server. These can be used, eg, in cases where a form on a WWW page the CGI script will take the data from the form, a database.

C
A well-known excellent string- and regular-expression-handling language.

ODBC
The Open Database Connectivity API. This defines a standard API for accessing databases. ODBC has taken several years to spread from being Microsoft specific to being shipped with UNIX operating systems, such as Solaris 2.5. ODBC is more complex than DBI!

Except, rather perversely, the platform ODBC originally sprang from, Windows/Window95/NT!

DBD
DataBase Driver

happen
Now, this seems quite obvious, but if I had a dollar for each time...

this
Assuming you're running a Bourne-compatible shell.

Bunce
Tim Bunce is the driving force behind the DBI initiative.

handle
Although this may be limited by the underlying database technology.