

Using Perl with the Informix Dynamic Server

Develop powerful applications for IDS

Skill Level: Intermediate

[Daniel Hebert](#)
Software Engineer
IBM

01 Mar 2007

This tutorial takes you through the process of setting up the Perl Database Interface (DBI) to work with the IBM® Informix® Dynamic Server (IDS). See examples of how to exercise a variety of DBI features and learn how to configure CPAN module installation, driver installation, script automation, data structures, pitfalls, and Perl interaction with IDS.

Section 1. Before you start

You must have IDS installed. This tutorial was designed using version 1.0, but later versions are expected to behave similarly.

About this tutorial

This tutorial walks you through the process of setting up the Perl DBI to work with IDS -- from start to finish -- and includes examples on how to exercise a variety of the DBI's features and learn about configuring CPAN module installation, driver installation, script automation, data structures, pitfalls, and Perl interaction with IDS. The tutorial assumes basic knowledge of Perl and builds on that. It also assumes you have IDS 10.00 running, but other versions can be expected to behave similarly. It is important to note that this tutorial is not intended to teach optimal Perl coding techniques, Perl 'style', or best Perl practices.

Objectives

The objective is to develop powerful applications for IDS quickly and easily using the capabilities of Perl and available tools.

Prerequisites

You must have a working knowledge of how to install and set up a simple database in IDS. Beginner programming skills are assumed. Prior Perl knowledge is helpful but not required. The code given in this tutorial shows you how to get set up.

System requirements

This tutorial uses Windows, but can be applied to any system that supports IDS and Perl (such as UNIX setups).

Section 2. Introduction

The first task at hand is to get a successful compilation of Perl on your machine. If you do not need any special features, refer to the [Resources](#) section of this document to obtain a binary from ActiveState. Alternatively, Google-search for a Perl binary that suits your needs. The ActivePerl distribution is offered for a variety of platforms and also features the standard Windows installer to save you time and aggravation.

Install Perl from the source in order to customize your install and support some features or variations not ready-made from a binary. This can be a daunting task due to the variety of compilers used, Windows' filenaming convention, and other issues that can crop up during installation. This article does not attempt to cover every compiler or possible installation scenario, nor does it comment on which compiler is "better". If you have special development needs, I recommend you compile from the source. See the [Resources](#) section for a link to the Perl Web site for more information.

Our second task is to configure Comprehensive Perl Archive Network (CPAN) module installation and driver installation. This is where the DBI and Open Database Connectivity (ODBC) become part of your Perl environment. We'll conclude with a demonstration and a discussion on how to use the DBI to communicate with IDS. By the end of this tutorial, you should be able to write your own custom interfaces to

interact with your Informix server quickly and easily.

Why the Perl DBI?

It is worth noting that Perl, as a "Web language", is both fortunate and cursed. Its abilities in CGI, which helped earn Perl this "claim to fame", have also contributed to the stereotypes as to its uses. The fact is, Perl is a robust, object-oriented language capable of:

- Writing graphical applications (platform independent)
- Capable of advanced shell scripting beyond what you would ever want to write in a UNIX shell script
- Capable of working with Web sites outside of the infamous CGI forms
- Automating virtually anything

Working with databases

Perl succeeds at targeting many programming cultures because it gives you the freedom to write code in whichever way you prefer. For this reason, this article does *not* judge proper Perl style. When this tutorial was written, Perl version 5.8.8 was the stable release version. Perl is still in the process of being developed.

The Perl DBI was written and is being developed by Tim Bunce. It is an interface that allows Perl programmers to quickly and easily develop full-featured database applications. For more information, refer to the [Resources](#) section.

Section 3. Before you start

You must set up Perl in your PATH variable. If you've installed Perl from ActiveState, this should already be taken care of for you on Windows. UNIX users may need to modify their environment variables to specify the Perl interpreter they'd like to use because many flavors of Unix come with Perl 5. If the following configuration instructions do not work for you, check your path and access to the Perl interpreter by typing **perl -v** on the command line. This should print the default interpreter being used.

Configuring CPAN and retrieving the DBI/ODBC modules

You must perform some installation work before you can code for your database.

You can find a listing of modules, developers, and mailing lists at [The CPAN Search Site](#). These modules are usually not included with your Perl installation, but may be useful. We will use some of them in this article. Perl comes with an application called CPAN that allows automated installation and access to these modules. Perl Package Manager (PPM), the other choice utility, comes with the ActiveState Perl binary, and is the preferred option for installations. Both installation methods for Perl modules are shown in this section:

I'm using CPAN installation:

If you're working with a version of Perl that is compiled from source or is otherwise unable to access PPM, you'll need to use the CPAN command-line program for module installation. Enter the **perl -MCPAN -e** shell into your command window. The CPAN application prompts you to configure its access to modules and to helper utilities, such as applications that decompress GZIP and TAR files. This is important because most of the modules you'll use are distributed in this format.

If you do not have any UNIX utilities available to you, a quick Web search will lead you to freeware to decompress these files. There are commercial compression/decompression packages that may do this as well, but without direct, standard command-line access to their specific decompression abilities (such as GZIP or TAR), you have to install these modules manually. See [CPAN FAQ's](#) for details on performing manual installations of Perl modules. Be warned, you may need several modules (including dependencies), so it is a much better to install a capable utility and point to it during your CPAN initialization.

Accidentally skipped the setup? No problem. Enter **o conf init** into your CPAN command line. If you have already set up your CPAN installation and need to make changes, you can edit a simple .pm file. Locate the '**Config.pm**' file in your CPAN directory and skip to the lines you need. For instance, my Config.pm file is in C:.

After you've gone through the CPAN setup, you're ready to start installing modules. Within the CPAN program, run the **install DBI** command followed by **install DBD::ODBC** command. Both of these modules are gzipped/tar'd. If you've set up your CPAN correctly, the rest of the installation process is automated.

Note: You can use the [Informix driver](#) in place of the ODBC module. It is more complicated to set up than DBD::ODBC, but is tailored to Informix. If you go this route, thoroughly read the documentation provided by the Web address and the driver's included README files.

I'm using PPM installation:

If you'd rather not use the CPAN module to install other modules, and you've installed the ActiveState Perl binary, you can enter **ppm** at the command line. A GUI comes up and allows you to see the modules you can upgrade, search for modules

to install, and more. Right-click and select the modules you need to install. Then, click the green arrow icon at the top to "Run marked actions." This is an easy way to get started fast and avoid the problems of the CPAN route. *If you happen to run into issues with the decompression or compilation of modules at any point, please check the CPAN section above, but PPM delivers precompiled modules to save you time and effort.*

What can I do if CPAN doesn't work after recompiling or reinstalling Perl?

Some uninstallers do not delete all of the Perl remnants after removal, including CPAN information. The intention may be to prevent the loss of modules you've accumulated over time. The assumption is that you are just setting up Perl for the first time and haven't downloaded any modules, so we aren't worried about destroying what's there. If this is true for you, do a hard disk search for ".cpan" to help you locate the old installation. You may want to back-up the folder first before you remove it manually, and then try setting up CPAN (possibly reinstalling Perl) again.

Section 4. Connecting to IDS

Making a simple connection to IDS

By this step, you should have successfully installed Perl 5.8.8 (or later) and the required modules.

Warning: DBI was not considered thread-safe when this tutorial was written.

Do not consider any module thread-safe unless you trust the author's guarantee, if any is given. In addition, Perl threads do not behave like threads in other languages. I do not recommend that you use them with the DBI in a production environment. In addition, the DBI does not allow threading on the same connection handle to a database. If you go this route, you need multiple connection handles, which may require an edit to your code or driver. Test them at your own risk.

This tutorial assumes you have successfully installed IDS 10.00 or later. At this point, you can start your database server if you have not done so already and make your first connection from Perl code. Open Notepad or your preferred editor. Perl does not need a "shebang" at the top of your code in Windows as long as your environment variables are set correctly. UNIX users may need to supply the path to the Perl interpreter at the top (for instance, `#!/usr/bin/perl`). Listing 1 is an example of the code necessary to make a simple connection to IDS:

Listing 1: How to connect

```
1: use strict;
2: require DBI;
3:
4: sub show_available_drivers_and_DSNs
5: {
6:
7:   my @drivers = DBI->available_drivers;
8:   print join(", ", @drivers), "\n";
9:   my @DSNS = DBI->data_sources('ODBC');
10:   for each my $d (@DSNS)
11:   {
12:     print "$d\n";
13:   }
14:
15:
16:
17: my $connection = "Host=TheComputerName;DSN=dan_IDSPhoenix";
18: my $connection2 = "DSN=dan_IDSPhoenix";
19:
20: my $dbhhandle = DBI->connect("dbi:ODBC:$connection2", "", "",
    RaiseError => 0, AutoCommit => 0)
21: or die("Can't connect, error msg: $DBI::errstr ");
22: &show_available_drivers_and_DSNs();
23: $dbhhandle->disconnect;
```

The first thing we did was enter `use strict` at the top of this code. One of Perl's notable features is that it lets you do many things quickly and easily, without the compiler harassing you for your “bad habits”, such as not scoping your variables. But these habits *are* bad, and in the interest of removing all ambiguity from the code, add this line to your Perl programs. Waiting until you're finished writing your scripts to do this could mean you end up with a long list of changes and waste a lot of time incorporating them.

In line 2, we add in the DBI functionality. In lines 4 through 15, we define our function to show the available drivers. For simplicity, the code here is shown as one file, but you'll want to get into the habit of sending these functions to your own packages and reusing them later. Line 7 and 9 show two self-explanatory functions from our newly installed interface, the DBI. If all goes well, you will see your available drivers followed by DSNs that can use ODBC.

Lines 17 and 18 are two possible connection strings; you're only going to use one. In line 20, we assemble the connection string by prepending “dbi:ODBC:” to the usual connection string you might use in other database applications. The two empty quoted fields would hold your username and password, if required. This is followed by a hash of attributes that allow you to control whether to automatically report errors, turn transaction support on or off, and more. You'll see how to do this later in this tutorial. This connection will be accessible through our database handle variable, “\$dbhhandle”. This variable is not required to be named “\$dbhhandle”, but it's easy to remember. Adopting a naming convention in your code is recommended. Finally, we call our driver/DSN print function and interpret our code with `perl`

filename.pl at a DOS command prompt. If all goes well, you should see your available drivers and DSNs that can use ODBC.

What should I do if the program hangs or nothing happens?

First, kill the process and then check your connection string syntax. Make sure the server started. Then, test the desired DSN connection to IDS by going to **Control Panel > Administrative Tools > Data Sources (ODBC) > Configure > Connection**.

Section 5. Interacting with IDS

After connecting: A basic program example

Now that you have a connection, it is important to become comfortable with basic database interaction. Simply build on the previous example by excluding the **disconnect** command.

Listing 2: Select statements

```
my $sql = qq!SELECT MAX(ID) FROM store!;  
$sthandle = $dbh->prepare($sql);  
(($sthandle) ? ($sthandle->execute()) :  
  (print $dbh->errstr . " is sql prepare error"));
```

We start with a straightforward select query from an imaginary table called “store”. You should modify it to accommodate your own tables. The first step is to assign your query to any variable. These variable names are meaningless to the DBI and can be anything you choose. Then, you would run the prepare method on your database connection handle that you defined earlier (on line 20) and assign it a variable to hold the return value, which is its own handle. Analyzing this return value will inform our next step. If an error occurs, the DBI places this error in `$dbh->errstr`. We will discuss better ways of handling errors later.

For ease of reading, we use the shorthand syntax for “if-then-else” in Perl. Read the last line as,

"If the prepare method returned positively, run the execute method on the returned handle, else print the error value that was automatically placed into `$dbh->errstr`."

Listing 3: The do function

```
my $drop_table = qq!DROP TABLE store!;  
($dbh->do($drop_table) ) ? (print "Successful drop: $drop_table") : ();
```

The *do* method combines the prepare and execute steps for non-SELECT SQL statements. It is slower than a prepare/execute combination because it recreates statement handles at each pass. If you find yourself in a loop that runs many times, save yourself the overhead of the *do* method and code a *prepare/execute* instead.

Listing 4: Fetching data

```
$bindcolsql = qq!SELECT name, balance, id FROM store!;  
$sth = $dbh->prepare($bindcolsql);  
$sth->execute();  
  
$sth->bind_columns($name,$balance,$id);  
  
while($sth->fetch)  
print "Bindcolumns:      Name: $name,      Id: $id";
```

There are several ways to fetch data. Some are more memory intensive than others. Listing 5 shows binding columns. You must correlate the columns with the order in your SELECT statement, for example *name* is column one, *balance* is column two, and so on. You may be notified of an error if you do not code the same number of references as there are columns. The fetch loop returns one row at a time per iteration. For example, **fetchall_arrayref** returns the least amount of calls for data but requires the most memory. DBI does have a performance measuring tool, and the topic itself deserves its own article. I will speak more to this topic later.

Listing 5: Placeholders

```
my $company_update_sql = qq!UPDATE store SET balance = ? WHERE id = ?!;  
...  
$sth = $dbh->prepare($company_update_sql);  
for ($k = 1; $k <= $level; $k++)  
...  
$sth->execute($balance, $companies[$flag][1]);
```

The advantage of placeholders is clear; you do not have to hard code all of your queries, and you also benefit from a performance boost. When using placeholders, the query plan is calculated only once and the values are substituted at execution instead of generating a new plan for every new value in the query. Placeholders give your application the ability to handle dynamic sets of data. Automated reports based on data mining, Web interaction with a data set, and many more functions become

quick and easy with the combined inherent strengths of Perl and IDS.

Listing 6: BLOBs and CLOBs

```

my $insertblob = qq!INSERT INTO item2 VALUES (1234,
'Desk', FILETOBLOB('screen.bmp',
'client'))!;
my $getblob = qq!SELECT LOTOFILE(screenshot, ?,
'client') FROM item2!;
...
$dbhandle = $dbh->prepare($insertblob);
($sth->execute()) :
    (print $dbh->errstr . " is insert blob prepare error");
...
$dbhandle = $dbh->prepare($getblob);
($sth->execute($blobfilename)) :
    (print $dbh->errstr . " is get blob prepare error");
...
$sth->bind_columns($blobtransfer);
while($sth->fetch)

```

Managing Character Large Objects (CLOBs) and Binary Large Objects (BLOBs) requires slightly more effort than a simple `SELECT` statement. Listing 6 shows how to insert and retrieve a BLOB from IDS using Perl and the DBI. We use `FILETOBLOB` to insert the BLOB into the database and `LOTOFILE` to retrieve the data. You'll notice we use `LOTOFILE` with a placeholder to copy the BLOB. You can do this to avoid getting "Out of memory!" errors from Perl. We perform the *prepare* and *execute* functions and then fetch the data as usual. Dealing with CLOBs simply requires changing the `FILETOBLOB` to `FILETOCLOB`.

If you do not want to use `LOTOFILE`, you must create some manual memory restrictions. Do **not** use the `dump_results` method for data transfer, although it might be tempting to use this method instead of fetching. For more information on the reasoning behind using the fetching method, refer to [CPAN Search Site: DBI](#). Also, for more information on `LOTOFILE`, check your Informix documentation and the [Resources](#) section for links to online information.

Listing 7: Enabling transaction support, checking for errors

```

my $dbh = eval DBI->connect("dbi:ODBC:$connection_string",
    "", "", RaiseError => 1, AutoCommit => 0);
...
eval(($balance < 0) ? ($dbh->rollback()) : ($dbh->commit()));
if($@)

print "$dbh->errstr error has occurred.";

```

Enabling transaction support and checking for errors requires a slight change to your

connection code. Before, we were individually testing all the return values, but there is a better, faster way to do this. We must set two variables -- *RaiseError* and *AutoCommit*. By turning *RaiseError* on, you can check the special variable "\$@" for any error flags raised by code wrapped in an *eval*. If you do not use this *eval* analysis when *RaiseError* is set to "1", the program throws an exception and dies.

This comes with the benefit of not only notifying of errors that occur during database interaction but also with your own Perl code. Instead of testing a handle specifically for a positive return value, we can wrap an *eval* around our code and appropriately deal with errors. *Eval* is very inexpensive to process and a preferred way to check for errors, as it will give better performance than individually testing handles. In addition, the DBI does feature detailed tracing capability to give you information on the driver in use and the DBI itself. For more information, see [The CPAN Search Site: Tracing](#).

Also, you may want to look up `prepare_cached` in the DBI documentation for queries you repeat often, such as during database transactions. For demonstration purposes, it is kept simple here.

Note: Database transaction support will fail unless your database has logging enabled. You will receive a "Driver Not Capable" error if you try `$dbh->rollback()` and `$dbh->commit()` without this. Log into IDS' dbaccess program and enable logging to resolve this issue.

Section 6. Optimization commentary

High-performance needs require optimization. Unfortunately, aside from general best practices, it is impossible to advise on how to optimize your specific application. Programming responsibly and reducing the overhead that goes along with communicating with a database is a good start, such as not creating handles when you don't have to. The DBI does have a nice utility to assist in giving informed statistics on application performance. See [CPAN Search Site: DBI Profile](#) for related documentation.

Need more performance information and advanced performance tips? Start with [DBI AdvancedTalk](#).

Section 7. Final notes on running the DBI

It is true that you could spend weeks optimizing and rewriting a Perl program to squeeze out every last millisecond of performance. It's also true that Perl is an interpreted language and therefore is not going to run faster than a language which compiles to machine code. Try to keep these factors in perspective as they are beyond your control: DBI implementation, driver choice and implementation, network latency, machine capability, and the version of your database. Set clear and realistic goals for your project from the beginning to avoid inconsistent error handling techniques and poor modular design. If you are new to programming for IDS in Perl, it is a good idea to run basic tests of the DBI functions that you plan to use for your project. Keep in mind that different drivers implement things differently. IDS, Perl, the DBI, and specific drivers are still actively being developed to accommodate evolving technology.

Resources

- ["Perl 5.8.8"](#) (Larry Wall, Copyright 1987-2006)
- ["ActiveState Perl, Dynamic Tools for Dynamic Languages"](#) (ActiveState Software, Inc., Copyright 2007)
- ["Downloading the Latest Version of Perl"](#) (Tom Christiansen, O'Reilly Media, Inc., 2006)
- ["About the Perl DBI"](#) (Tim Bunce, Copyright 2003-2006)
- ["DBI"](#) (Tim Bunce, Copyright 2003-2006)
- ["DBI- Database Independent Interface for Perl"](#) (Tim Bunce, Copyright 2003-2006)
- ["DBI: Profile - Performance Profiling and Benchmarking for the DBI"](#) (Tim Bunce, Copyright 2003-2006)
- ["DBI_AdvancedTalk"](#) (Tim Bunce, Copyright 2003-2006)
- ["IBM Informix Built-In Datablade Modules User's Guide"](#) (IBM Corporation Copyright 1996,2004,2005)
- ["Perl Database Programming"](#) (Brent Michalski. John Wiley & Sons, Copyright 2003 ISBN:0764549561)
- [Informix Dynamic Server Documentation](#), Copyright 2005

About the author

Daniel Hebert

Daniel Hebert has his B.S. and graduate level experience in Computer Science from Rensselaer Polytechnic Institute. He has spent two years as a teaching assistant for a computer science department Perl course. Dan is a Software Engineer for IBM, primarily working with Informix Dynamic Server (IDS).

Trademarks

This is the first trademark attribution statement.

This is the second trademark attribution statement.