

Perl DBI Examples

Contents

- [Basics](#)
 - [Making the connection](#)
 - [Options](#)
 - [Issuing SQL](#)
- [Intermediate](#)
 - [SELECT statements](#)
- [Advanced](#)
 - [Transactions](#)
 - [Oracle stored procedures](#)
- [Notes](#)
- [Other Resources](#)

The purpose of this document is to provide some examples for using the [Perl DBI modules](#) to access databases. The motivation for this is to relieve some of the FAQ traffic on the [DBI mailing list](#).

Over the course of this document, we will build a robust DBI program to access an Oracle database. We will start with DBI basics, then introduce concepts to improve performance and reliability.

Basics

The first thing we need to do is build and install DBI. Instructions for this are in the DBI INSTALL document. Then we will need to build a database driver, or [DBD](#). Instructions for building the DBDs are also included with each package. As with most Perl modules, installing DBI/DBD is usually as easy as

```
localhost:/opt/src/perl_modules/DBI-1.13$ perl Makefile.PL &&  
make && make test && make install
```

After DBI and a DBD are installed properly, you can get (a lot) more information by issuing:

```
localhost:~$ perldoc DBI
```

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.

```
use strict;
use DBI;

my $dbh = DBI->connect( 'dbi:Oracle:orcl',
                       'jeffrey',
                       'jeffspassword',
                       );
```

The connect string above takes three arguments: a data source name, a username, and a password. The DSN is in the form `dbi:DriverName:instance`. But how do we know if the connect succeeded or not? First, `connect` will return a true value on success, untrue otherwise. Second, DBI will place an error message in the package variable `$DBI::errstr`.

```
use strict;
use DBI;

my $dbh = DBI->connect( 'dbi:Oracle:orcl',
                       'jeffrey',
                       'jeffspassword',
                       ) || die "Database connection not made:
$DBI::errstr";
$dbh->disconnect();
```

Using the `disconnect()` method will avoid the error "Database handle destroyed without explicit disconnect".

Options

The `connect()` method can take a hash of options. Often-used options include: `AutoCommit`, which when true will automatically commit database transactions; `RaiseError`, which tells DBI to croak `$DBI::errstr` upon errors; and `PrintError`, which tells DBI to warn `$DBI::errstr`.

In this program, we will want to use transactions, so we will turn `AutoCommit` off, `RaiseError` on, and leave `PrintError` at its default of on.

```
use strict;
use DBI;

my $dbh = DBI->connect( 'dbi:Oracle:orcl',
                      'jeffrey',
                      'jeffspassword',
                      {
                        RaiseError => 1,
                        AutoCommit => 0
                      }
                    ) || die "Database connection not made:
$DBI::errstr";
$dbh->disconnect();
```

Note that setting `AutoCommit` off with a database that doesn't support transactions will result in a fatal error.

Issuing SQL

Now we are ready to do something useful with our database. There are two ways to get an SQL statement to your database. For queries which return rows, such as `SELECT`, we will use the `prepare` method. For other queries, such as `CREATE` and `DELETE`, we will use the `do` method. Let's stick to the latter for now and move on to the former later.

This program will create an employee table in the database.

```
use strict;
use DBI;
```

```

my $dbh = DBI->connect( 'dbi:Oracle:orcl',
                       'jeffrey',
                       'jeffspassword',
                       {
                           RaiseError => 1,
                           AutoCommit => 0
                       }
                       ) || die "Database connection not made:
$dbh->errstr";

my $sql = qq{ CREATE TABLE employees ( id INTEGER NOT NULL,
                                       name VARCHAR2(128),
                                       title VARCHAR2(128),
                                       phone CHAR(8)
                                       ) };

$dbh->do( $sql );

$dbh->disconnect();

```

Intermediate

We have seen how to connect to the database, detect errors, and issue simple SQL statements. Now let's move on to some more useful code.

SELECT Statements

The `SELECT` statement is probably the most often used statement in SQL. To use `SELECT`, we will first prepare the statement and then execute it. In the following code, the `$sth` is the statement handle, which we will use to access the result of the `SELECT`.

```

use strict;
use DBI;

my $dbh = DBI->connect( 'dbi:Oracle:orcl',
                       'jeffrey',
                       'jeffspassword',
                       {

```

```

        RaiseError => 1,
        AutoCommit => 0
    }
) || die "Database connection not made:
$DBI::errstr";

my $sql = qq{ SELECT * FROM employees };
my $sth = $dbh->prepare( $sql );
$sth->execute();

$dbh->disconnect();

```

The listing above will cause the database to make an execution plan for the statement, then execute that statement. It doesn't actually do anything with the rows returned. In the next listing, we use `bind_columns` to get the records out of the database. `bind_columns` binds each column to a scalar reference. When `fetch` is called, those scalars are filled with the values from the database.

```

use strict;
use DBI;

my $dbh = DBI->connect( 'dbi:Oracle:orcl',
    'jeffrey',
    'jeffspassword',
    {
        RaiseError => 1,
        AutoCommit => 0
    }
) || die "Database connection not made:
$DBI::errstr";

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

my( $id, $name, $title, $phone );
$sth->bind_columns( undef, \$id, \$name, \$title, \$phone );

while( $sth->fetch() ) {
    print "$name, $title, $phone\n";
}

```

```

}

$sth->finish();
$dbh->disconnect();

```

That's a nice program for printing out a company phone book, but how about a WHERE clause? We will use `bind_param` to prepare an SQL statement one time, and execute it several times very quickly.

```

use strict;
use DBI qw(:sql_types);

my $dbh = DBI->connect( 'dbi:Oracle:orcl',
                      'jeffrey',
                      'jeffspassword',
                      {
                        RaiseError => 1,
                        AutoCommit => 0
                      }
                    ) || die "Database connection not made:
$DBI::errstr";

my @names = ( "Larry%", "Tim%", "Randal%", "Doug%" );

my $sql = qq{ SELECT id, name, title, phone FROM employees WHERE
name LIKE ? };
my $sth = $dbh->prepare( $sql );

for( @names ) {
    $sth->bind_param( 1, $_, SQL_VARCHAR );
    $sth->execute();

    my( $id, $name, $title, $phone );
    $sth->bind_columns( undef, \$id, \$name, \$title, \$phone );

    while( $sth->fetch() ) {
        print "$name, $title, $phone\n";
    }
}

```

```
$sth->finish();
$dbh->disconnect();
```

Advanced

Transactions

So far, we haven't done anything that would require transactions, but if we need to issue UPDATE or DELETE statements, we will want to use them. The best way to implement robust transactions with DBI, according to the DBI documentation, is to use `eval{...}` blocks to trap errors, then use `commit` or `rollback` to finish the transaction. That is what we will do in the following listings.

This program loads four records into our database.

```
use strict;
use DBI qw(:sql_types);

my $dbh = DBI->connect( 'dbi:Oracle:orcl',
                      'jeffrey',
                      'jeffspassword',
                      {
                        RaiseError => 1,
                        AutoCommit => 0
                      }
                    ) || die "Database connection not made:
$dbh::errstr";

my @records = (
    [ 0, "Larry Wall",      "Perl Author",    "555-
0101" ],
    [ 1, "Tim Bunce",      "DBI Author",    "555-
0202" ],
    [ 2, "Randal Schwartz", "Guy at Large", "555-
0303" ],
    [ 3, "Doug MacEachern", "Apache Man",    "555-
0404" ]
);
```

```

my $sql = qq{ INSERT INTO employees VALUES ( ?, ?, ?, ? ) };
my $sth = $dbh->prepare( $sql );

for( @records ) {
    eval {
        $sth->bind_param( 1, @$_->[0], SQL_INTEGER );
        $sth->bind_param( 2, @$_->[1], SQL_VARCHAR );
        $sth->bind_param( 3, @$_->[2], SQL_VARCHAR );
        $sth->bind_param( 4, @$_->[3], SQL_VARCHAR );
        $sth->execute();
        $dbh->commit();
    };

    if( $@ ) {
        warn "Database error: $DBI::errstr\n";
        $dbh->rollback(); #just die if rollback is failing
    }
}

$sth->finish();
$dbh->disconnect();

```

Calling Oracle stored procedures

One question I am frequently asked, in person and via the DBI users' mailing list, is how to call stored procedures using DBD::Oracle. Here I will give examples of varying complexity.

The program calls a stored procedure with one in parameter and no return value. We assume that the procedure does not call `commit`. Note the use of positional placeholders in this program. Also note the use of the `eval` block: if your Oracle procedure raises an exception, it will be translated into `die` in your Perl program, and the error message will be placed in `$@` and `$DBI::errstr`.

```

use strict;
use DBI;

my $dbh = DBI->connect(
    'dbi:Oracle:orcl',

```

```

    'jeffrey',
    'jeffspassword',
    {
        RaiseError => 1,
        AutoCommit => 0
    }
) || die "Database connection not made: $DBI::errstr";

eval {
    my $func = $dbh->prepare(q{
        BEGIN
            jwb_function(
                parameter1_in => :parameter1
            );
        END;
    });

    $func->bind_param(":parameter1", 'Bunce'); #positional
    placeholders are handy!
    $func->execute;

    $dbh->commit;
};

if( $@ ) {
    warn "Execution of stored procedure failed: $DBI::errstr\n";
    $dbh->rollback;
}

$dbh->disconnect;

```

The next program calls a stored function with a return value. To return a value from a function, we bind the placeholder using `bind_param_inout`. When using this method, we must tell `DBD::Oracle` how many bytes to allocate for the return value.

```

use strict;
use DBI;

my $dbh = DBI->connect(

```

```

'dbi:Oracle:orcl',
'jeffrey',
'jeffspassword',
{
    RaiseError => 1,
    AutoCommit => 0
}
) || die "Database connection not made: $DBI::errstr";

my $rv; #holds the return value from Oracle stored procedure
eval {
    my $func = $dbh->prepare(q{
        BEGIN
            :rv := jwb_function(
                parameter1_in => :parameter1
            );
        END;
    });

    $func->bind_param(":parameter1", 'Bunce');
    $func->bind_param_inout(":rv", \$rv, 6);
    $func->execute;

    $dbh->commit;
};

if( $@ ) {
    warn "Execution of stored procedure failed: $DBI::errstr\n";
    $dbh->rollback;
}

print "Execution of stored procedure returned $rv\n";

$dbh->disconnect;

```

Notes

1. The use of `finish` in these examples is not strictly necessary. You should call it if you are done with the statement handle but not with your program.
2. Always use `strict`.

Other Resources

- [DBI home page](#)
- [How to use mod_perl and Apache::DBI](#)

This document Copyright [Jeffrey William Baker](#). Last modified 11 November 2003



Other documents on this site include:

- [Application Performance using DBI and mod_perl](#)
- [Resumé of Jeffrey William Baker](#)