JOHNS HOPKINS UNIVERSITY

Zanvyl Krieger School of Arts and Sciences
Advanced Academic Programs

410.712.81 Advanced Practical Computer Concepts For Bioinformatics

Topic: Perl DBI

Instructor: Joshua Orvis

What is DBI?

The Database Interface is a perl module that makes it relatively easy to connect to and perform operations within a database.

It has built-in **drivers** for connecting to many different database vendors such as MySQL, PostgreSQL, Oracle, etc., and includes even file-based databases like BerkeleyDB and SQLite.

From the DBI documentation:

> *The DBI "dispatches" the method calls to the appropriate driver for actual execution. The DBI is also responsible for the dynamic loading of drivers, error checking and handling, providing default implementations for methods, and many other non-database specific duties.*

This separation between DBI and the underlying drivers provides a wonderfully useful abstraction layer, so you can write your perl code in a generic way and, if you eventually change database vendors, you only have to change the line of code that connects to the database.

Database drivers

The perl DBI provides the programming interface that is implemented by database drivers (DBDs). The most common of these DBDs come pre-packaged with modern perl distributions, but you can always find more of them on CPAN.

The following Perl code will show you what DBDs you have on your system already:

```
#!/usr/bin/perl -w

use strict;

use DBI;

for my $driver ( DBI->available_drivers ) {
    print "DBD::$driver\n";
}
```

On my laptop, that yielded the following modules by default:

DBD::DBM	DBD::Proxy
DBD::ExampleP	**DBD::SQLite**
DBD::File	DBD::Sponge
DBD::Gofer	**DBD::mysql**

Getting documentation

You can get generic DBI documentation using perldoc:

```
$ perldoc DBI
```

But you can get driver-specific documentation by calling for it directly:

```
$ perldoc DBD::mysql
```

Connecting to a database

For general usage, this is really the only place where your code will change based on the database vendor you're using. Generically, the connection string looks like this:

```
use DBI;
my $dbh = DBI->connect($data_source, $user, $pass, \%attr);
```

There's a lot of configuration within that though. Also notice that you don't have to 'use' the specific DBD you want to connect with, it's autoloaded by the DBI module.

The $data_source above is the biggest source of variation within database vendors. It is here you should check the perldoc for your vendor to connect, but for MySQL a connection looks like this:

```
my $dsn = 'DBI:mysql:database=jorvis_chado;host=localhost';
My $dbh = DBI->connect($dsn, $user, $pass, { RaiseError => 1, PrintError => 1 });
```

That's a section of code you can copy and use in your own applications after changing the *database* and *host* values.

The hashref at the end is where you can pass a lot of different options, but these are two very common ones. By default, if anything errors in your database while using DBI it will silently carry on the next operation. By setting both these options, you're telling DBI to both raise an exception when an error is encountered and report what it was. (**You may not want to report the error on a web application, since it might expose private/insecure data**.)

I recognize database coding can be some dry stuff, however useful, so I hope you enjoy these random cartoons/pictures to break it up a bit. If you'd like to see your favorites in future lectures, e-mail them to me.

I DON'T ALWAYS TEST MY CODE
BUT WHEN I DO I DO IT IN PRODUCTION
memegenerator.net

Security note

You should have noticed that the connection string contains quite a lot of information that you might not want to hard-code into your application and expose to others. User names, passwords, etc. all must be passed to the DBI *connect* method, but in practice you don't want to embed them directly in your code. Also, if you have a multi-user application you'll need to have a way to pass them dynamically depending on which user is logged in.

There are a few solutions here. A simple one is to create a standard INI file, which is just a set of key/value pairs in a plane text file with some section headers. You can keep your connection information in that file and make sure it's not exported if you send others your code. (See the perl module Config::IniFiles for simple usage of this module.)

This is also beneficial because you might have dozens of scripts connecting to the same database, and if you house the connection information within a single file you only need to change that file if something about your database connection changes.

Conventions

You'll notice some strange abbreviations for variable names by most users when using the DBI module. For some reason, these names have become so expected it's almost surprising when someone doesn't follow them. Most of them involve “handles”, which are data structures containing information about a connection or active set of data. I'll use these abbreviations throughout the class, and there are slight variations, but they are:

$dbh = A database handle, returned by a call to the connect() method.

$dsh = A database statement handle, returned by a call to the prepare() method. (this is often also referred to as a $sth (statement handle).

$row = Representing a row of data, this is returned when looping and calling any of the 'fetch' methods (see later slides).

Selecting data - overview

To issue a SELECT statement you need to do the following generic steps:

1. connect to a database
2. prepare a SQL statement
3. execute the SQL statement
4. iterate through your results
5. 'finish' your statement
6. disconnect from the database

Connecting to the database is often the most time-expensive step of this, so it's common practice to connect at the beginning of your script and hold the connection open until you no longer need it. It is usually very bad practice to repeatedly connect to your database within the same script.

Let's do an example of a simple case.

Selecting data - example

In this example, we perform all of the steps mentioned on the previous slide.

```
my $dsn = 'DBI:mysql:database=jorvis_chado;host=localhost';
my $opts = { RaiseError => 1, PrintError => 1 };

my $dbh = DBI->connect($dsn, 'jorvis', 'secretpassword', $opts);

my $qry = qq{
    SELECT feature_id, uniquename
      FROM feature
};

my $dsh = $dbh->prepare($qry);
   $dsh->execute();

## iterate through the results
while ( my $row = $dsh->fetchrow_hashref ) {
    print "$$row{feature_id}\t$$row{uniquename}\n";
}

## clean up
$dsh->finish();
$dbh->disconnect();
```

Let's look at the new query definition/execution/iteration portion of the script a little closer.

Smart.

Selecting data - example

First we have the definition of our SQL statement using perl's qq function so that it's easier to read across multiple lines. This is especially useful as your queries get larger and more complex:

```
my $qry = qq{
    SELECT feature_id, uniquename
      FROM feature
};
```

Next we tell DBI to prepare our statement for use, then to execute it. We'll see later that using this construct instead of other (possibly shorter) alternatives will benefit us in the long run, especially if you're passing values in a WHERE clause or executing a query multiple times with different parameters.

```
my $dsh = $dbh->prepare($qry);
   $dsh->execute();
```

Once we execute the query our statement handle can be used to iterate through the results. Here I used fetchrow_hashref(), which places the data from each row into a hash reference:

```
while ( my $row = $dsh->fetchrow_hashref ) {
    print "$$row{feature_id}\t$$row{uniquename}\n";
}
```

Fetching rows

Once you've executed a statement you can fetch rows (usually in a while loop) from the handle using three different methods, depending on how you want to access the columns in your data.

$dsh->fetchrow_array()

Perhaps the most basic, each call to this method returns an array for that row where each element is a database column, in the order you defined them in your SELECT statement.

$dsh->fetchrow_arrayref()

The same as above but as an array reference rather than an array. This saves ever-so-slightly on memory and execution time.

$dsh->fetchrow_hashref()

I use this one almost exclusively – it returns a hash reference for each row where the keys are the column names from your SELECT statement and the value is the value for that column. I find this the easiest to maintain since you don't have to remember the column numbers or go back and change them if you add a column to your query at a later date.

A *while* loop is typically used to iterate with these methods because they all return undef when there are no more rows to return, which automatically ends the while loop.

Conditional selects and placeholders

Our previous example was a simple SELECT with no WHERE clause. If you need to pass values to filter on using a WHERE clause this is tempting:

```
    my $qry = qq{
        SELECT feature_id, uniquename
          FROM feature
         WHERE uniquename = '$some_name'
    };
```

DON'T DO IT. Don't. Just say no. Bad Things will happen (ask Sony.)

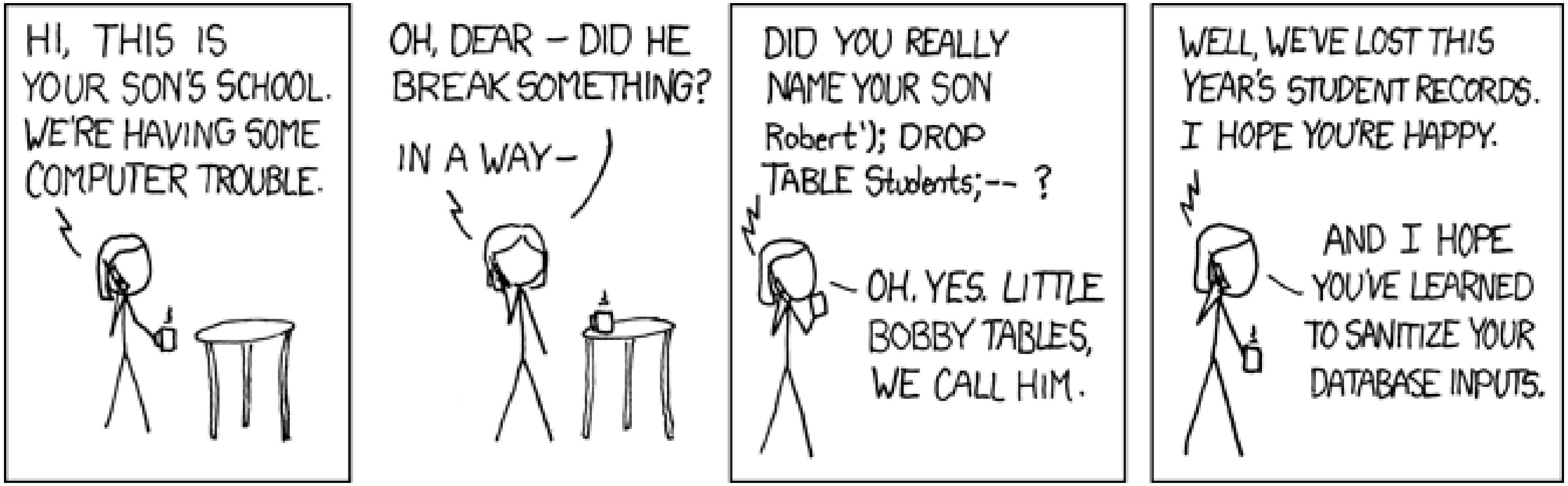
For ANY instance where you need to put a value, either hard-coded, entered by a user or from a CGI parameter, you should use *placeholders*. That is, put a ? mark there and then pass the value you want to go there as an argument to the *execute* method. If you have two ? marks, just pass two values in the same order to execute(), and so on. For example:

```
    my $qry = qq{
        SELECT feature_id, uniquename
          FROM feature
         WHERE uniquename = ?
    };
    my $dsh = $dbh->prepare($qry);
       $dsh->execute( $some_name );
```

Failure to do this results in a major security vulnerability known as SQL injection, where a Bad Guy could end your SQL statement and insert one of his own ...

SQL injection

This has been an unbelievably serious and embarrassing problem for many companies, who have had private data stolen from them because they failed to take simple steps like this one. The execute method 'scrubs' or 'sanitizes' the values passed, adding escape characters as needed, to make sure this doesn't happen.


HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Re-execution of SELECTs

If you want to query data for multiple with specific names, for example, you'll want to re-execute the same prepared statement handle but with a different values in your WHERE clause. Using placeholders and reusing the same prepared statement handle makes this much, much faster than redoing all those steps each time. Here's an example (imagine you had an array with thousands of names, and you know your query will return exactly 0 or 1 rows):

```
my @names = qw( AF87234.1 FR98123.1 JJ009238.2 );

my $qry = qq{ SELECT * FROM feature WHERE uniquename = ? };
my $dsh = $dbh->prepare($qry);

for my $name ( @names ) {
    $dsh->execute( $name );
    $row = $dsh->fetchrow_hashref || next;

    print "$$row{feature_id}\t" . length($$row{residues}) . "\n";
}

$dsh->finish();
```

Within the *for* loop we re-execute the statement handle, each time passing a different value for our WHERE clause arguments.

Inserting, deleting data

Doing INSERTs and DELETEs follows these same steps, except there are no resulting rows to iterate once they're complete. Here are examples of both:

```
my $qry = qq{
    INSERT INTO featureprop
        (featureprop_id, feature_id, type_id, value, rank)
    VALUES (?, ?, ?, ?, ?)
}
my $dsh = $dbh->prepare($qry);
   $dsh->execute( $fp_id, $f_id, 56, 'aldolase', 0 );


$qry = qq{
    DELETE FROM feature
    WHERE feature_id = ?
};

$dsh = $dbh->prepare($qry);
$dsh->execute( $fid );

$dsh->finish();
```

If you have lots of inserts and deletes to do (perhaps you're reading data from a file) you can use placeholders and execute your statement handle within a loop just as we did for selects on the previous slide.

do() method

For the other commands you might want to do, such as creating tables, adding indexes, etc., you can use the database handle's do() method, which just executes any command you pass directly.

```
$dsh->do("CREATE DATABASE jorvis_chado");

$dsh->do("DROP TABLE foo");

$dsh->do("CREATE INDEX id_index ON feature (id) USING BTREE");
```

Lies!

TO OPEN
PUSH HERE
LIES!
KRAFT
Macaroni & Cheese
DINNER
The Cheesiest
ORIGINAL FLAVOR
COOKING INSTRUCTIONS

Transactions

Transactions are a way to group data manipulation statements together so that if any of them fail they the others can be undone. If your database supports transactions, you can use these methods.

First, the DBI auto-commits all statements by default. You can turn this off in your script if you plan to use transactions by doing this:

```
    $dbh->{AutoCommit} = 0;
```

Then, you can use the following (aptly-named) methods to commit or rollback transactions:

```
    $dbh->commit() = 0;

    $dbh->rollback() = 0;
```

In practice, if you connect to the database like the examples so far have shown, the script on the next slide is a good strategy for handling transactions.   Generically, you wrap your commands in an eval block to trap any errors that occur. If you find any, rollback, otherwise commit.

Transactions, an example

```
my $dsn = 'DBI:mysql:database=jorvis_chado;host=localhost';
my $opts = { RaiseError => 1, PrintError => 1, AutoCommit => 0 };

my $dbh = DBI->connect($dsn, 'jorvis', 'nevermore', $opts);

## catch any errors
eval {

    $dbh->do($some_sql);
    $dbh->do($some_other_sql);

    ## if you get this far, there weren't any errors, so commit
    $dbh->commit();
};

## rollback if we found any errors (which are stored in $@)
if ($@) {
    print "rolling back transaction: $@";
    $dbh->rollback();
}

# close connection
$dbh->disconnect();
```

Getting more information

This covers the key functionalities in DBI so you can get started with working with databases in Perl. As in everything else, there are a lot of shortcuts and alternative methods you can use. See the perldoc for both DBI and your specific database drivers for much, much more information.