practical perl

Database Modeling with Class::DBI

by Adam Turoff

Adam is a consultant who specializes in using Perl to manage big data. He is a long-time Perl Monger, a technical editor for *The Perl Review*, and a frequent presenter at Perl conferences.

ziggy@panix.com

Database programming is often boring and tedious. Even with modules like DBI, using a database routinely involves writing nearly the same code over and over again. But it doesn't need to be that way. Class::DBI removes most, if not all, of the tedium and makes it easier to use a database than to avoid it.

I started a new job recently, and now I spend about an hour every day commuting on the train. This affords me a nice stretch of time to catch up on my reading, and I now regularly read one or two books per week. However, as I plow through my library, I'm starting to forget some of the books I've read recently.

Of course, there are many ways to solve this admittedly minor annoyance. I could whip up a quick little Web-based application in Perl, and store the list of books I've read recently in some database. But that strikes me as an approach that is simple, obvious, and wrong. The last thing I want to do, actually, is spend my time at home writing the same kind of database-centric Web application that I build at work.

So I looked for a simple, lazy solution. I just wanted something to jog my memory, and for a while I kept a list of book titles on a 3x5 card. But a list of book titles is a little limiting. A simple list of titles doesn't give me a whole lot of room to keep summaries of what I've read, or write notes on things I want to remember.

As this simple 3x5 card system started to get unwieldy, I started to think again about writing a Perl program to help me manage this information. After all, I could use SQLite as a database, which is about as painless as relational databases can possibly get, especially for quick hacks like this.

Even with SQLite, I wasn't looking forward to writing the Perl code to add, insert, and update records in a database. Again. I've done it for years, and while it's not that hard, it is repetitive and somewhat tedious. So I looked into Tony Bowden's Class::DBI module, which makes almost all of that pain just go away.

Step 1: Creating a Database

If you have been writing Perl code to use a database for even a short period of time, you know the story by heart:

- Create an empty database to hold your data.
- Connect to the database with the DBI module.
- Prepare SQL SELECT statements, and retrieve rows from the database as necessary.
- Process those rows, one by one.
- Occasionally prepare SQL statements to insert, update, or delete rows in the database.

The first step in writing a database-centric application is to set up the database. Class::DBI cannot help here, so I still need to do this the old-fashioned way with SQL CREATE TABLE statements.

To start, I only care about maintaining a list of books. Books are written by authors, so I will need two tables to start—one for books and one for authors. This will allow me to find all books written by a single author, but it will only let me track the primary author for any one book. Not a perfect design, but good enough to start—I'm not trying to replace the Library of Congress here.

The SQL statements to create these tables in SQLite look like this:

```
CREATE TABLE author (
    authorid INTEGER PRIMARY KEY,
    first,
    middle,
    last);

CREATE TABLE book (
    bookid INTEGER PRIMARY KEY,
    author,
    title,
    subtitle,
    pubyear,
    edition,
    isbn,
    format);
```

The first thing to note is that SQLite is a typeless database, so all columns will be stored as strings of arbitrary size. This means that it is not an error to store a value like "199x" in the pubyear column, something which would normally store only integer values. So SQLite will never raise an error when interpreting the value "199x" as an integer, because all values are always strings, and there are never any type errors to worry about. Nor will SQLite raise an error when it tries to store a 1024-character string in a 32-character field—it will always store the full 1024 characters.

The one minor exception to this rule is the case when the first field in a table is declared as the INTEGER PRIMARY KEY for that table. In this case, SQLite will automatically assign a unique integer ID value for every row added to this table. This will be the record ID for each record.

With these two CREATE TABLE statements, creating an SQLite database is a breeze. SQLite stores its databases as regular files on the file system, so if I have access to this file, I have access to this database—no database user IDs and passwords to worry about! If these two SQL statements are stored in a file called create.sql, I can create the database like this:

```
sqlite  library.db   <  create.sql
```

(Or I could write a small Perl program and use DBI calls to create the database. But this way is quicker and easier—something lazy and impatient programmers everywhere will appreciate.)

Step 2: Using Class::DBI

Now that there is an empty database, I need Perl code to add, edit, update, and delete rows. This is where things start to get very boring, very quickly.

Normally, I could start to write a Perl program by loading the DBI module, creating a database handle, and issuing SQL statements against that database handle, mixing Perl and SQL code all along the way. Alternatively, I could write a series of modules that hide these low-level operations to access my database, keeping all of the SQL code neatly isolated in one place.

However, the only things that differ between this application and the database applications I wrote last week, last month, or last year are:

- How to connect to the database
- The tables in this database
- The connections between these tables
- The columns in these tables

Class::DBI enables me to access a database just by specifying these four key pieces of information. In exchange, it provides an object-oriented interface for accessing the database, and hides all of the low-level details, like constructing SQL statements. For simple applications, I can forget that there is a SQL database lurking underneath (after I issue the CREATE TABLE statements, that is).

Using Class::DBI is simple. Very simple. It starts with a module that defines the first piece of information in my little application—how to connect to the database:

```
package MyLibrary::DBI;
use base "Class::DBI";
MyLibrary::DBI->connection("dbi:SQLite:library.db", "", "");
```

Next, I need to define one module for each of the two tables in my database. Each of these modules should declare three things: the name of the table in the database, the columns in that table, and the relationship between this table and other tables in this database. Here are those two module definitions:

```
package MyLibrary::Author;
use base "MyLibrary::DBI";

MyLibrary::Author->table("author");
MyLibrary::Author->columns(All => qw(authorid first middle last));
MyLibrary::Author->has_many(books => "MyLibrary::Book");

package MyLibrary::Book;
use base "MyLibrary::DBI";

MyLibrary::Book->table("book");
MyLibrary::Book->columns(
  All => qw(bookid author title subtitle pubyear edition isbn format)
);
MyLibrary::Book->has_a(author => "MyLibrary::Author");
```

And that is it. By writing these three trivial modules, Class::DBI will provide the guts of a database application to manage my list of books. All of the tedious, repetitive database management code is automatically provided by Class::DBI.

Step 3: Programming with Class::DBI

But what does Class::DBI actually do?

Class::DBI works by defining the common behaviors that generic database application share. It does all of the hard work, but it doesn't know how to use this particular database. To make it work in a real application, I need to plug in the specifics—where to find the database, and the structure inside that database. That's what the three modules do: customize a general-purpose framework for database applications into the framework I need for my database application today. Each of those customizations is made through inheritance (those important use-base clauses peppered about).

That's fine, but how do I use the modules I just created? It's quite simple, really. To create a new row in the author table, create a new MyLibrary::Author object. If I use the MyLibrary::Author module to search for an author record, I will get an object that represents a row in the author table. I can make changes to one of these objects, and commit those changes back to the database at a later time.

Creating a new record is easy. Just create a new object, as if it were any other Perl module:

```
# Create an empty author object,
# and an empty record in the database
my $author1 = new MyLibrary::Author;

# Create another object/row, and specify
# values for two fields
my $author2 = new MyLibrary::Author({
        first => "Isaac",
        last  => "Asimmov"
});
```

When I created the MyLibrary::Author module, I defined three important fields in the author table: first, middle, and last. Class::DBI makes these fields in MyLibrary::Author objects, and provides methods to get and set the values of these fields. Changes to these objects can be sent back to the database using the update method, and records can be deleted using the delete method on these objects:

```
## This was a blank record in the database.
## Add some values to it, and write it back.
$author1->first("Edgar");
$author1->middle("Rice");
$author1->last("Burroughs");
$author1->update();

## Wait a second — I don't have any books by this author!
$author1->delete();

## Oops! "Asimov" was misspelled
$author2->last("Asimov");
$author2->update();
```

Of course, book records are just as easy to create, update, and delete.

Class::DBI and Database Design

There is an additional wrinkle, though. Most relational databases are "normalized" to some degree. Class::DBI knows about normalization and understands that most databases are at least somewhat normalized. In my simple design for the MyLibrary database, each book record is uniquely identified by the value in its bookid field, and each author record is uniquely identified by its authorid field. Because books have authors, the author field in the book table stores one of these authorid values, which can be used to find the author of a book. (A fuller discussion of database normalization is beyond the scope of this article, but this is generally how it works in practice.)

Unique record IDs are such a common feature in relational database designs that Class::DBI automatically assumes that the first field in a table contains the record ID for that table—a safe assumption, since this practice is quite common.

Class::DBI has two main methods to describe the common relationships between tables: has_a and has_many. In this simple database design, an author has many books, and a book has a single author. (Remember, this is an admittedly simplistic model of bibliographic data.) With the MyLibrary::Book->has_a(author=> "MyLibrary::Author") declaration, I've told Class::DBI a few things:

- The author field in the book contains a unique ID from the author table.
- When I select a record from the book table, the value isn't a plain number, it is a reference to the author table. Class::DBI should use that value to instantiate the MyLibrary::Author object that corresponds to that ID value.
- If I assign MyLibrary::Author object to an author field in a MyLibrary::Book object, Class::DBI should use the ID for that author, not some other value for that author.

In this database, a book can have one author, but an author can have many books. This means that many records in the book table can share a single author value. This relationship is stated in the MyLibrary::Author->has_many(books=>"MyLibrary ::Book") declaration, and it adds a method to the MyLibrary::Author module named add_to_books. This method allows me to create or edit a book record, and set that record's author field to the unique ID of this MyLibrary::Author record:

```
## First, create an author.
my $author = new MyLibrary::Author({
        first => "Isaac",
        last  => "Asimov"
});

## Create a new, standalone book record.
my $book1 = new MyLibrary::Book({
        title => "Foundation"
});
my $book2 = new MyLibrary::Book({
        title => "Foundation and Empire"
});

## Three ways to set the author of a book.
$book1->author($author);
$author->add_to_books($book2);

## Create a new book, and set its author.
$author->add_to_books({
    title => "Second Foundation"
});
```

There are many, many more methods that Class::DBI provides for manipulating data in relational databases. Some of the more important ones are:

- retrieve(): fetch an object given its unique ID value
- search(): fetch many objects given a list of fields to match
- find_or_create(): find an existing record that matches the fields specified, or create a new record with those values

Conclusion

This article just barely scratches the surface of using Class::DBI. This module has many more features, and it supports the standard databases such as MySQL, PostgreSQL, Oracle, and DB2. While this sample program demonstrates that the combination of SQLite and Class::DBI makes a quick database hack simple and easy, Class::DBI is a heavily used, robust package for simplifying all kinds of database programs.

For good measure, here is the full source of the MyLibrary module that uses Class::DBI to create an interface to my library database, and the addbook script to add a book to this database. Note that most of the work is not in managing the database, but in managing user input (editing a text file with $EDITOR and processing the result).

MYLIBRARY.PM.

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

package MyLibrary;

package MyLibrary::DBI;
use base "Class::DBI";
MyLibrary::DBI->connection("dbi:SQLite:library.db", "", "");

package MyLibrary::Author;
use base "MyLibrary::DBI";

MyLibrary::Author->table("author");
MyLibrary::Author->columns(All => qw(authorid first middle last));
MyLibrary::Author->has_many(books => "MyLibrary::Book");

package MyLibrary::Book;
use base "MyLibrary::DBI";

MyLibrary::Book->table("book");
MyLibrary::Book->columns(
  All => qw(bookid author title subtitle pubyear edition isbn format)
);
MyLibrary::Book->has_a(author => "MyLibrary::Author");

1;
```

ADDBOOK.

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

use strict;
use MyLibrary;

sub get_input {
      my @template = @_;
      my $tmpfile = "/tmp/library.data.$$";

      ## Write out the template.
      open (my $fh, ">$tmpfile");
      print $fh @template;
      close($fh);

      ## Edit it...
      $ENV{EDITOR} ||= "vi";
      system("$ENV{EDITOR} $tmpfile");

      ## Read it back.
      open($fh, $tmpfile);
```

```
    my @data = &lt;$fh>;
    close($fh);

    ## Remove the temp file, and return the user data.
    unlink $tmpfile;
    return (@data);
}

sub process_input {
    my @lines = @_;
    chomp(@lines);

    my ($group, %author, %book);

    foreach my $line (@lines) {
        ## Group name/value pairs into Author and Book blocks.
        if ($line =~ m/^(\w+)$/) {
            $group = $1
        } elsif ($line =~ m/^\s+(\w+):\s*(.*)$/) {
            ## Ignore fields that weren’t set.
            next unless $2;

            if ($group eq ”Author”) {
                $author{lc($1)} = $2;
            } elsif ($group eq ”Book”) {
                $book{lc($1)} = $2;
            }
        }
    }

    return \%author, \%book;
}

my ($author, $book) = process_input(get_input(&lt;DATA>));

my $author_rec = MyLibrary::Author->find_or_create($author);
my $book_rec   = MyLibrary::Book->find_or_create($book);

$book_rec->author($author_rec);
$book_rec->update();

__DATA__
Author
      First:
     Middle:
       Last:
Book
      Title:
   Subtitle:
    Edition:
       ISBN:
    PubYear:
     Format:
```