Class::DBI

how to avoid writing code

Jacinta Richardson
<jarich@perltraining.com.au>

Perl Training Australia

SQL

- Who here likes writing SQL?
- I mean really likes writing SQL?
- Who'd solve the following problem in SQL?

```
  SEND
+MORE
-----
MONEY
```

There's always someone...

```
SELECT
    (S.X * 1000 + E.X * 100 + N.X * 10 + D.X) as SEND,
    (M.X * 1000 + Oh.X * 100 + R.X * 10 + E.X) as MORE,
    (M.X * 10000 + Oh.X * 1000 + N.X * 100 + E.X * 10 + Y.X) as MONEY
FROM
    Dx M
    LEFT JOIN Dx S ON (M.X != S.X)
    LEFT JOIN Dx E ON ( (M.X != E.X) AND (S.X != E.X) )
    LEFT JOIN Dx Oh ON ( (M.X !=Oh.X) AND (S.X !=Oh.X) AND (E.X !=Oh.X) )
    LEFT JOIN Dx N ON ( (M.X != N.X) AND (S.X != N.X) AND (E.X != N.X)
           AND (Oh.X != N.X) )
    LEFT JOIN Dx R ON ( (M.X != R.X) AND (S.X != R.X) AND (E.X != R.X)
           AND (Oh.X != R.X) AND (N.X != R.X) )
    LEFT JOIN Dx D ON ( (M.X != D.X) AND (S.X != D.X) AND (E.X != D.X)
           AND (Oh.X != D.X) AND (N.X != D.X) AND (R.X != D.X) )
    LEFT JOIN Dx Y ON ( (M.X != Y.X) AND (S.X != Y.X) AND (E.X != Y.X)
           AND (Oh.X != Y.X) AND (N.X != Y.X) AND (R.X != Y.X)
           AND (D.X != Y.X) )
    LEFT JOIN Cx C0 ON ( C0.X = floor( (D.X + E.X) / 10 ) )
    LEFT JOIN Cx C1 ON ( C1.X = floor( (N.X + R.X + C0.X) / 10 ) )
    LEFT JOIN Cx C2 ON ( C2.X = floor( (E.X + Oh.X + C1.X) / 10 ) )
    LEFT JOIN Cx C3 ON ( C3.X = floor( (S.X + M.X + C2.X) / 10 ) )
WHERE
    ( M.X != 0 ) AND
    ( S.X != 0 ) AND
    (                C3.X    ) = M.X  AND
    MOD( S.X + M.X + C2.X, 10) = Oh.X  AND
    MOD( E.X +Oh.X + C1.X, 10) = N.X  AND
    MOD( N.X + R.X + C0.X, 10) = E.X   AND
    MOD( D.X + E.X       , 10) = Y.X
;
```

Wouldn't it be great...

- If you didn't have to worry about database connections
- If you could just create an object and it would be populated from data in the db.
- If you could just make changes to an object and they'd magically happen to the db
- If you didn't have to write SQL all through your code AND you didn't have to do more work to avoid doing so?

That's why we have Class::DBI

- Class::DBI gives us all of this
- And more....
- Class::DBI handles the relational aspect of a relational database
- It knows that tables can have a one-to-many, or many-to-one, or a maybe-one relationship with other tables.
- It even allows you to add triggers.

Our example

- The following example is based on a recent real-life project (simplified)
- The tables aren't sensibly normalised.
- We have two interesting tables:
  - Pictures
    - A list of all the interesting information kept about each image
  - Categories
    - Categories that the pictures fit into. This table maps the picture id to a category name, rather than to a category id.

An example

- First we create a subclass of Class::DBI to set up our own connection, environment variables etc.

```
package Pix::DBI;
use strict;
use base 'Class::DBI';

Pix::DBI->connection($dsn,
                     $username,
                     $password,
                     {
                         RaiseError => 1,
                         ShowErrorStatement => 1
                     }
);
```

An example

- Then we create our Picture table's class.

```
package Pix::Pictures;
use strict;
use base 'Pix::DBI';

my @fields = qw/image_number image_title prepared_by
                description/;

__PACKAGE__->table('PIX.PICTURES');
__PACKAGE__->columns(Primary => "image_number")
__PACKAGE__->columns(All => @fields);

__PACKAGE__->has_many(
             categories => 'Pix::Categories',
             {order_by =>  'category_name'}
);
```

has_many

- The has_many method is how we tell Class::DBI that this table is related to another table in a one-to-many relationship.
- This means that each picture may have zero or more categories it fits into.
- When a has_many relationship is specified on a table, a has_a (or has_many) relationship must also be specified on the corresponding table

An example

- Our categories class would look like this:

```
package Pix::Categories;
use strict;
use base 'Pix::DBI';

my @fields = qw/image_number category_name/;

__PACKAGE__->table('PIX.CATEGORIES');
# Assumes first column is primary
__PACKAGE__->columns(All => @fields);

__PACKAGE__->has_a(
              image_number => 'Pix::Pictures',
);
```

Example use

- Once we've created our class definitions we can use these classes simply and easily.

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

my $picture = Pix::Pictures->retrieve( $image_number );

printf("%s (%d) fits into the following categories:\n",
      $picture->image_title(), $picture->image_number);

print join("\n", $picture->categories());
```

Creating a new object

- Object (and thus database row) creation is easy:

```
my $new =  Pix::Pictures->create(
                    {
                        image_number => undef,
                        image_title  => $title,
                        prepared_by  => $prepared,
                        description  => $desc,
                    }
);

# required if you're using transactions
$new->dbi_commit();
```

Updating and deleting

- To make changes to the database we just change our object

```
$new->image_title($new_title);
$new->description($new_description);

 # commit if required
 $new->dbi_commit();
```

- To delete an object we just call delete

```
 $picture->delete();
```

- This cascades deletes where required.

Adding/deleting categories

- To add values to our relations we do the following:

```
$new->add_to_categories(
                {
                    category_name => $category,
                }
);
```

- To delete all categories we can write:

```
$new->categories()->delete_all();
```

Searches

- Class::DBI provides very simple `retrieve`/`retrieve_all` and `search`/`search_like` methods

```
my $picture = Pix::Pictures->retrieve($image_number);

# What, no ordering?
my @pictures = Pix::Pictures->retrieve_all();

my @owned = Pix::Pictures->search(
                  prepared_by => $me,
                  {order_by => image_number}
);

my @elephants = Pix::Pictures->search_like(
                  title => '%elephant%'
);
```

Better searches

- You can also create your own searches (with a little SQL) by specifying the WHERE ....

```
# in Pix::Pictures.pm

__PACKAGE__->add_constructor(
        all_start_end => qq{
                image_number >= ? AND
                image_number <= ?
                ORDER BY image_number
        },
);

# Later, in our code:
my @between = Pix:Pictures->all_start_end($start, $end);
```

Better searches

- Or by writing the whole statement....

```
# in Pix::Pictures.pm

# search_all like retrieve_all, but with ordering.
__PACKAGE__->set_sql(all =>
        qq{
                SELECT __ESSENTIAL__
                FROM   __TABLE__
                ORDER BY image_number
        }
);

# Later, in our code:
my @all = Pix::Pictures->search_all();
```

Problems with Class::DBI

- Class::DBI isn't as well documented as I'd like. In quite a few cases it's a case of try-it-and-see.
- Class::DBI attempts to guess the primary key if it's not provided at creation time. For MySQL this works a treat, but for some databases Class::DBI has problems.

More problems

- Class::DBI uses "magic" (stringfying) structures in the relation objects. Eg:

```
# get first category
my $category = $picture->categories()->first();

print Dumper $category:

$VAR1 = bless ({
          'image_number' => bless ({
                                     'image_number' => 42,
                                     'image_title' => 'fred',
                                     'prepared_by' => 'June',
                                     'description' => '',
                                   }, 'Pix::Pictures'),
          'category_name' => 'cat 1'
          }, Pix::Categories );
```

Further resources

- Maypole is built on top of Class::DBI

http://maypole.perl.org/

- Turnkey is build on top of Class::DBI

http://sumo.genetics.ucla.edu/turnkey/index.php?n=Main.HomePage

- The wiki:

http://www.class-dbi.com/cgi-bin/wiki/index.cgi

- CPAN

http://search.cpan.org/~tmtm/Class-DBI-0.96/lib/Class/DBI.pm