

*Copyright © 2005
Randal L. Schwartz
Stonehenge Consulting Services, Inc.
+1 (503) 777 0095
<http://www.stonehenge.com/merlyn/>*

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

Introduction to Class::DBI

Randal L. Schwartz

www.stonehenge.com/merlyn/

Version 1.4 at 2 Mar 05

Overview

- *Why Class::DBI?*
- *Requirements for Class::DBI*
- *An example database*
- *Details*
- *Add-on packages in the CPAN*

Why Class::DBI?

- *Perl's DBI abstraction layer removes most of the tedium of SQL client layers*
- *But you still have to write a lot of mind-numbing repetitive SQL for most apps*
- *There's a lot to get right*
- *Writing SQL requires a brainshift*
- *SQL doesn't provide easy code re-use like Perl's objects*

How does CDBI solve this?

- *Each table in your database becomes a class*
- *Each row is represented in memory by an instance of that class*
- *Columns become member variables, fetched and stored with named accessors*
- *Foreign key access fetches objects in the referenced table*

Too simple?

- *Yeah, it sounds simple*
- *But it means that for 95% of your database operations, you don't write a single line of SQL*
- *And the remaining 5% can be isolated into a class definition, so it still doesn't appear sprinkled in your code*
- *You can still do one-offs directly*

Efficiency matters!

- *Column data is lazy-loaded—only the parts you actually reference*
- *But you can also say that everything gets loaded at once for a different kind of optimization*
- *SQL is “prepared” and some servers cache that as well*

Requirements

- *CDBI presumes a single primary key*
- *CDBI can be coaxed into using multiple-column primary keys*
- *But foreign keys linking to that table are only partially supported*
- *Thus, you can generally wrap CDBI around most decent database designs*
- *But there'll be some that won't work*

An example database

- *Consider a simple three table DB*
- *Films (with title, year)*
- *People (with name, birthdate)*
- *Roles (what a Person did for a Film, including alternate name and type of role: actor, director, etc.)*

Keeping it simple for CDBI

- *Give each of the three tables an autoincrementing “id” integer column*
- *Can't key on name anyway (not unique)*
- *“Gone in 60 seconds” has both a 1974 and a 2000 version for example*
- *Or there's the classic Beau Geste (3 versions)*
- *This permits renaming the film as well*

Sample table definitions

- *Film:*

```
CREATE TABLE film (  
  id INTEGER PRIMARY KEY, [generally need some sequencing info too]  
  name VARCHAR(128),  
  year INTEGER  
)
```

- *Person:*

```
CREATE TABLE person (  
  id INTEGER PRIMARY KEY, [generally need some sequencing info too]  
  name VARCHAR(128),  
  birthdate DATE  
)
```

- *Role:*

```
CREATE TABLE role (  
  id INTEGER PRIMARY KEY, [generally need some sequencing info too]  
  film INTEGER REFERENCES film,  
  person INTEGER REFERENCES person,  
  job VARCHAR(128)  
)
```

Defining these for CDBI

- *Film:*

```
package MyDB::Film; use base MyDB;
__PACKAGE__->table('film');
__PACKAGE__->columns(All => qw(id name year));
```

- *Person:*

```
package MyDB::Person;
use base MyDB;
__PACKAGE__->table('person');
__PACKAGE__->columns(All => qw(id name birthdate));
```

- *Role:*

```
package MyDB::Role;
use base MyDB;
__PACKAGE__->table('role');
__PACKAGE__->columns(All => qw(id film person job));
__PACKAGE__->has_a(film => 'MyDB::Film');
__PACKAGE__->has_a(person => 'MyDB::Person');
```

But what's in MyDB?

- *Defining my base class:*

```
package MyDB;  
use base Class::DBI;  
__PACKAGE__->connection($dsn, $user, $password);
```

- *This establishes the connection to the database*
- *\$dsn is a standard DBI connection string*
- *You can also stick anything else here that the rest of the table classes need*
- *Yes, if you'd rather define those tables automatically, see `Class::DBI::Loader`*

Populating the database

- *Create a film:*

```
use MyDB::Film;  
my $modern_sixty = MyDB::Film->create({  
    name => 'Gone in 60 seconds',  
    year => 2000,  
});
```

- *At this point, a new object is created in memory*

- *The “id” will be unique:*

```
print $modern_sixty->id, "\n";
```

- *We can update the title:*

```
$modern_sixty->name('Gone in 60 Seconds');  
$modern_sixty->update;
```

- *Updates are in memory until ->update is called*

Now let's add some actors

- *Create two stars:*

```
use MyDB::Person;
my $cage = MyDB::Person->create({name => 'Nicolas Cage'});
my $jolie = MyDB::Person->create({name => 'Angelina Jolie'});
```

- *Put them into the movie:*

```
use MyDB::Role;
for my $person ($cage, $jolie) {
    MyDB::Role->create({
        person => $person,
        title => 'Actor',
        film => $modern_sixty,
    });
}
```

- *Let's put Angelina in a few more films:*

```
MyDB::Role->create({film => $_, person => $jolie, job => 'Actor'}) for
    MyDB::Film->create({name => 'Tomb Raider', year => 2001}),
    MyDB::Film->create({name => 'Girl, Interrupted', year => 1999}),
    MyDB::Film->create({name => 'Gia', year => 1998}),
    MyDB::Film->create({name => 'Hackers', year => 1995});
```

Basic queries

- *Get Angelina back from the database:*

```
my @angelinas = MyDB::Person->search(name => 'Angelina Jolie');  
die "Duplicate or missing Angelina Jolie" unless @angelinas == 1;
```

- *Show all films that start with a G:*

```
for my $film (MyDB::Film->search_like(name => 'G%')) {  
    printf "%20s (%d)\n", $film->name, $film->year;  
}
```

- *Dump the roles for those films:*

```
for my $film (MyDB::Film->search_like(name => 'G%')) {  
    printf "Film %s (%d):\n", $film->name, $film->year;  
    for my $role (MyDB::Role->search(film => $film) {  
        printf "  %s (%s)\n", $role->person->name, $role->job;  
    }  
}
```

Optimizations for many-to-many

- *Define people for a film, and films for a person:*

```
package MyDB::Film;  
__PACKAGE__->has_many(people => ['MyDB::Role' => 'person']);  
package MyDB::Person;  
__PACKAGE__->has_many(films => ['MyDB::Role' => 'film']);
```

- *Now we can bridge quickly:*

```
my @jolie_films = $jolie->films;  
my @modern_sixty_people = $modern_sixty->people;
```

- *These also define “add_to_...” methods*

```
$jolie->add_to_films({film =>  
  MyDB::Film->create({name => 'Original Sin', year => 2001})  
});
```

- *However, the extra columns in the linking table (here, ‘job’) are not set when add_to_ is used*

The gritty details

- *CDBI class methods*
- *CDBI table methods*
- *Setting up triggers and constraints*
- *CDBI table instance methods*
- *Relationships*
- *Defining additional SQL*
- *Controlling lazy population*

Class methods

- *connection(\$dsn, \$user, \$password):* define the database
- *table(\$tablename):* define the table within that database
- *sequence(\$sequence_name):* define a sequence to fetch for NULL primary key
- *db_Main:* return \$dbh for direct DBI operations on the underlying database

Table methods

- *create(\%values): create a new row, given the values*
- *If the primary key is omitted, it's fetched from a sequence*
- *find_or_create(\%values): find an existing row, or create if not found*
- *delete: delete the object instance*
- *delete also cascades through related recs*

Table methods for finding rows

- *retrieve(key_column => \$value): fetch a row by primary key*
- *retrieve(\$value): for single-column primary key*
- *retrieve_all: fetch every row in a table*
- *search(column => \$value, column => \$value, ...): search for exact match*
- *search_like(...): search with “LIKE”*

Triggers

- *Triggers are called when various actions occur to table-row objects*
- *Unrelated to database triggers, but they serve a similar Perl-side purpose*
- *Add a trigger:*
`__PACKAGE__->add_trigger($when => $coderef);`
- *Trigger names:*
`before_create, after_create, before_set_$column, after_set_$column,
before_update, after_update, before_delete, after_delete, select`

Constraints

- *Constraints are set up per column:*
`__PACKAGE__->add_constraint($column_name => $coderef);`
- *Constraints get called on each update to the column*
- *Constraints are passed the new value, the row object, the column name, and a hashref of all columns and new values*
- *Constraints must return true/false*

Normalization and validation

- *When any data is changed, `$object->normalize_column_values` is called*
- *Hash of new values is passed (and original object before change)*
- *`$object->validate_column_values`*
- *Default implementation calls all the `before_set_$column` triggers*
- *Define `$self->_croak` for exceptions*

Accessors

- *Accessors: ->get('column') and ->set('column', 'new_value')*
- *Easier to type: ->column, and ->column('new_value')*
- *Define altered accessor names:*

```
sub accessor_name {  
    my ($class, $column) = @_;  
    $column =~ s/^customer//;  
    return $column;  
}
```
- *Now you can use ->name instead of ->customername*

Update and autoupdate

- *\$row->update: flush changed values to the database*
- *\$row->autoupdate(\$enabled): set or clear auto updates*
- *\$row->dbi_commit, \$row->dbi_rollback: useful when update is off, to commit or roll back a transaction*

Relationships

- *has_a(column => 'table class')*:
establish foreign key (won't work for multi-column foreign key)
- *has_many(method => 'table class')*:
establish reverse foreign key (other class will be loaded if needed)
- *has_many(method => ['table class' => 'method'])*: *call method on mapped class*

“might_have” relationships

- *might_have(method => 'class name' => @fields)*
- *Causes @fields of class to be transparently mapped*
- *Example: Role using an alternate name:*

```
package MyDB::Role;
__PACKAGE__->might_have(alias => 'MyDB::Alias' => qw(known_as));
my ($r) = MyDB::Role->search(person => $jolie, film => $sixty_modern);
my $different_name = $r->known_as; # same as $r->alias->known_as;
```

Defining additional SQL

- *add_constructor(method => 'SQL')*
- *Turns into:*

```
SELECT __ESSENTIAL__  
FROM __TABLE__  
WHERE [your code]
```
- *Returns a list of matching objects when method is called*
- *Placeholders can be used:*

```
MyDB::Film->add_constructor(after_year => 'year > ?');  
...  
my @recent_films = MyDB::Film->after_year(2000);
```
- *Sent to prepare, so query plan is cached*

One-off SQL

- *retrieve_from_sql(\$where)*
- *Returns row objects satisfying where and order-by*
- *Better to put this into add_constructor when you can*

Controlling lazy population

- *Primary columns make up the key*
- *Essential columns are always fetched when the row object is created*
- *“Other” columns can be accessed as needed*
- *“TEMP” columns exist only in memory*

Defining column groups

- *\$table_class->columns(Primary => qw(id))*: define “id” as the primary key
- *\$table_class->columns(Essential => qw(name))*: always fetch the name
- *\$table_class->columns(Other => qw(address city state zip))*: others
- *\$table_class->columns(All => (id name address city state zip))*: ‘id’ is primary

CDBI add-ons

- *Class::DBI::Loader*
- *Class::DBI::AbstractSearch*

Class::DBI::Loader

- *Fetch table info from database*
- *Use that to set up CDBI classes:*

```
use Class::DBI::Loader;
my $loader = Class::DBI::Loader->new(
    dsn => $dsn, user => $user, password => $password,
    namespace => "MyDB");
# list of classes loaded is $loader->classes
# list of tables connected is $loader->tables
```
- *Also sets up index sequences automatically*
- *Can be coaxed into adding has_a and has_many relationships as well*

Class::DBI::AbstractSearch

- *Wrapper around SQL::Abstract*
- *Provides complex AND/OR/NOT queries with properly escaped values:*

```
my @election_year_films_beginning_with_g =
MyDB::Film->search_where({
    name => { 'LIKE', 'G%' },
    year => [2004, 2000, 1996, 1992, 1988, 1984, 1980, 1976],
});
```

For more information

- *perldoc* “*Class::DBI*”
- *CDBI* wiki at:
<http://www.class-dbi.com/cgi-bin/wiki/index.cgi>