A command line counter with database back-end using DBIx::Class

In this simple example we'll see how to use DBIx::Class (aka. DBIC) to create and use a database for a simple command-line counter script.

DBIx::Class is an ORM, an Object Relational Mapper. It tries to bridge the conceptual difference between the representation of the data in a Relational database where it is stored in tables, and the application where the data is stored in Objects.

In order to understand it you'll have to understand both objects and relational databases, though you don't necessarily need to know SQL, the language that is usually used to talk to relational databases. One of the key features of every ORM, and thus DBIC is that, it eliminates the need to use SQL. At least in the common cases.

The end product

In this example we'll build a command line counter that works like this:

```
$ perl dbic_counter.pl
Usage: dbic_counter.pl name

$ perl dbic_counter.pl foo
Creating new counter for foo
1

$ perl dbic_counter.pl foo
2

$ perl dbic_counter.pl bar
Creating new counter for bar
1

$ perl dbic_counter.pl foo
3
```

```
$ perl dbic_counter.pl foo
4

$ perl dbic_counter.pl bar
2
```

It expects a single parameter on the command line, the name of a counter. Every time we provide a name, the scripts displays the next number for that counter starting from 1. Apparently I even left in a debugging message showing when each counter is created.

The code representing the database

We have 3 files in the example:

The main script called dbic_counter.pl, the module representing the whole database (or schema) lib/MyCounter/Schema.pm,
and lib/MyCounter/Schema/Result/Counter.pm that defines and represents the 'counter' table.

The directory layout looks like this:

```
$ tree
.
├── dbic_counter.pl
└── lib
    └── MyCounter
        ├── Schema
        │   └── Result
        │       └── Counter.pm
        └── Schema.pm
```

The module representing the whole database (the schema) is quite simple: It sub-classed DBIx::Class::Schema and then calls the load_namespaces method on the current packages. (__PACKAGE__ always contains the current name-space in it, which is the name declared using the package keyword. See this article if you need some clarification with the names.)

lib/MyCounter/Schema.pm

```
1.  package MyCounter::Schema;
2.  use strict;
3.  use warnings;
4.  use base qw(DBIx::Class::Schema);
5.
6.  __PACKAGE__->load_namespaces();
7.
8.  1;
```

load_namespaces will load all the modules under the MyCounter::Schema::* name-space representing all the tables of the database. In our case there is only one such file.

The code of the MyCounter::Schema::Result::Counter module is in lib/MyCounter/Schema/Result/Counter.pm. It subclasses DBIx::Class::Core.

Then we declare the name of the table in the database (counter), the names of the columns (name and cnt). We also have declare one of the fields as primary_key. In many cases a special id field is created to be the primary key, but in our case we expect the name to be unique, so we can use that as a primary key.

```
1.  package MyCounter::Schema::Result::Counter;
2.  use strict;
3.  use warnings;
4.  use base qw(DBIx::Class::Core);
5.
6.  __PACKAGE__->table('counter');
7.  __PACKAGE__->add_columns(qw( name cnt ));
8.  __PACKAGE__->set_primary_key('name');
9.
10. 1;
```

That's how we describe the counter table.

The script using the database

Once we created the above two files representing the database we can see the script using them.

```
1.  use strict;
2.  use warnings;
3.  use 5.010;
4.
5.  use lib 'lib';
6.  use MyCounter::Schema;
7.
```

```
8.  my $name = shift or die "Usage: $0 name\n";
9.
10. my $file = 'dbix_counter.db';
11.
12. my $schema = MyCounter::Schema->connect("dbi:SQLite:$file");
13.
14. if (not -e $file) {
15.     $schema->deploy();
16. }
17.
18. my $counter = $schema->resultset('Counter')->find($name);
19. if (not $counter) {
20.     say "Creating new counter for $name";
21.     $counter = $schema->resultset('Counter')->create({
22.         name  => $name,
23.         cnt   => 0,
24.     });
25. }
26. $counter->cnt( $counter->cnt + 1);
27. say $counter->cnt;
28. $counter->update;
```

(The only reason we require version 5.010 here is because I am lazy and I wanted to use say instead of print.)

We start with use lib 'lib'; to adjust @INC to include the lib subdirectory and then we load MyCounter::Schema, the main module of the database. This module will in turn load the module representing the counter table.

my $schema = MyCounter::Schema->connect("dbi:SQLite:$file"); connects to the database. It is a bit like the connect method of DBI, but it returns an object representing the schema.

The comes the nice part. The deploy method of this object will create the actual tables in the database. Based on the description we provided in the modules describing the database, it will create SQL statements (CREATE TABLE ...) and will create the tables. We call it only if the file did not exist earlier as we don't want to try to create the tables again and again on every run.

In the next line

```
1.  my $counter = $schema->resultset('Counter')->find($name);
```

$schema->resultset('Counter') represents the counter table. The find method will search in the table for the given value in the primary key field (which is the name field in this case), and return and object representing the row.

The block in the if (not $counter) { statement is going to be executed once for every counter: The first time we use the counter. There is the left-over debugging statement declaring we are creating a new counter.

The create method on the object representing the counter table is basically the same as the INSERTSQL statement. It accepts a hash-reference with the key-value pairs of the field names and their initial values. After inserting the row in the database it also creates an object representing that row in the database. This object is assigned to $counter. This is exactly the same kind of object as was (or would have been) returned by the all method a few lines earlier.

So either way, after the block of the if-statement we have an object in $counter representing the row in the database where the name field equals to the value given by the user.

$counter->cnt($counter->cnt + 1); increments the field in the object - not yet changing the database.

$counter->update; writes the changes value to the database. Similar to the SQL UPDATE statement.

Prerequisites

For the above code to work one needs to install DBIx::Class, DBD::SQLite, and in order to have the deploymethod work we also need to install SQL::Translator

If we don't have SQL::Translator installed, the script will still start to run, but when we encounter the deploy call it will throw an exception:

```
## DBIx::Class::Storage::DBI::deployment_statements(): Can't deploy
without a ddl_dir or SQL::Translator >= 0.11016

(see DBIx::Class::Optional::Dependencies for details) at
dbic_counter.pl line 8
```

It will also create an empty file for the database and so if you run the script again (even if you already installed SQL::Translator), you'll get another exception:

```
DBIx::Class::Storage::DBI::_prepare_sth(): DBI Exception:
DBD::SQLite::db prepare_cached failed: no such table: counter

[for Statement "SELECT me.name [me.name], me.cnt FROM counter me WHERE
( name =

? )"] at ./dbic_counter.pl [dbic_counter.pl] line 18
```

Just remove the database file dbix_counter.db manually and run the script again. (after installing SQL::Translator)

The SQL schema

If you are comfortable with SQL, you might want to know what is going on behind the scenes.

Adding the following call to the main script:

```
1.  $schema->create_ddl_dir(['SQLite'],
2.                          '0.1',
3.                          './'
4.                          );
```

will create a file called MyCounter-Schema-0.1-SQLite.sql that will contain the SQL which can be used to create the database schema. The content of this file is this:

```
--
-- Created by SQL::Translator::Producer::SQLite
-- Created on Fri May  9 16:55:54 2014
--


BEGIN TRANSACTION;


--
-- Table: counter
--
DROP TABLE counter;


CREATE TABLE counter (
  name  NOT NULL,
  cnt  NOT NULL,
  PRIMARY KEY (name)
);


COMMIT;
```

In case you use a different tabase (MySQL, PostgreSQL, etc.) you can ask DBIx::Class to show the respective SQL statements.

Debugging - SQL statements

Adding the following line:

```
1.  $schema->storage->debug(1);
```

will tell DBIx::Class to print each SQL statement as it executes them.

If we run the script with a name that already started to count (foo) the output looks like this: (The number 5 is the regular output of our script) Here we can see a SELECT statement fetching the fields of table, and then an UPDATE statement saving the new value.

```
SELECT me.name, me.cnt FROM counter me WHERE ( me.name = ? ): 'foo'
5
UPDATE counter SET cnt = ? WHERE ( name = ? ): '5', 'foo'
```

If we run the script with a name that we have not used earlier (other) the output is slightly different. In addition to the SELECT and UPDATE statements we also have an INSERT statement that added the initial 0 value to the database.

```
SELECT me.name, me.cnt FROM counter me WHERE ( me.name = ? ): 'other'
Creating new counter for other
INSERT INTO counter ( cnt, name) VALUES ( ?, ? ): '0', 'other'
1
UPDATE counter SET cnt = ? WHERE ( name = ? ): '1', 'other'
```

Conclusion

That's it for now. We just saw one of the most simple example using DBIx::Class.