111% DBIx::Class

In dieser Ausgabe gibt es den zweiten Teil der Mini-Serie "100% DBIx::Class", in der es um ein paar Tricks geht, mit denen man die Möglichkeiten von `DBIx::Class` besser nutzen kann.

In dieser Folge wird es darum gehen wie man den Suchpfad für bestimmte Klassen ändern kann. Weiterhin geht es um so genannte "Virtuelle Views". Dabei kann man das Ergebnis einer Abfrage als "Tabelle" nehmen. Im letzten Abschnitt geht es um Profiling und wie man damit Statistiken über die Laufzeit von SQL-Befehlen erstellen kann.

load_namespaces

`load_namespaces` ist eine Alternative zu `load_classes` und ist dann nützlich wenn man eigene Result- oder ResultSet-Klassen hat, die nicht im Suchpfad liegen. Im Suchpfad liegen alle Klassen aus den `DBIx::Class::Result::*`- bzw. `DBIx::Class::ResultSet::*`-Namensräumen. Manchmal ist es aber erforderlich, dass die selbst geschriebenen Klassen außerhalb des Suchpfades liegen. Dann kommt `load_namespaces` ins Spiel.

Wie eigene ResultSet-Klassen geschrieben werden, wurde im letzten Teil von "100% DBIx::Class" gezeigt.

Hier drei Beispiele für die Verwendung von `load_namespaces`. Im ersten Beispiel liegen die Klassen alle unterhalb von `My::Schema`. Da die Klassen in `::Result::*` bzw. `::ResultSet::*` liegen - was die Standardnamen in `DBIx::Class` sind - braucht man `load_namespaces` keine Parameter zu übergeben.

```
# lädt My::Schema::Result::User,
# My::Schema::ResultSet::User etc.
My::Schema->load_namespaces;
```

Listing 1

Im zweiten Beispiel wird gezeigt, wie man von den typischen `DBIx::Class`-Namen abweichen kann. Die Result-Klassen liegen nicht in `::Result::*` und die ResultSet-Klassen nicht in `::ResultSet::*`, sondern in `::MyResults::*` bzw. in `::MyResultSets::*`. Aber auch diese Klassen liegen unterhalb von `My::Schema`.

```
# lädt My::Schema::MyResults::User,
# MySchema::MyResultSets::User etc.
My::Schema->load_namespaces(
  result_namespace => 'MyResults',
  resultset_namespace => 'MyResultSets',
);
```

Listing 2

Das dritte Beispiel zeigt, wie man vorgeht, wenn die Klassen auch nicht mehr unter `My::Schema` liegen. Soll `DBIx::Class` mitgeteilt werden, dass der Namensraum ein voll qualifizierter Name ist, muss ein `+` vorangestellt werden.

```
My::Schema->load_namespaces(
  result_namespace => '+Namespace::Results',
  resultset_namespace => '+Namespace::Sets',
);
```

Listing 3

Virtuelle Views

Eine View ist eine logische Relation in einem Datenbanksystem. Diese logische Relation wird über eine gespeicherte Abfrage definiert. In der Anwendung kann die View wie eine "normale" Tabelle behandelt werden und immer wenn diese View verwendet wird, wird die dahinterliegende Abfrage ausgeführt. Die gespeicherte Abfrage ist also schon eine Vorselektion der Daten.

Dies wird häufig verwendet, wenn man Nutzern nur bestimmte Daten zur Verfügung stellen will, weil ihnen z.B. Administrationsrechte fehlen. Eine andere Anwendung ist das Zusammenstellen von Informationen aus verschiedenen Tabellen.

Die folgenden Code-Fragmente zeigen, wie man mit `DBIx::Class` so eine View erstellen kann. Als erstes muss eine Klasse erstellt werden, die die View darstellt (Lising 4). Diese Klasse muss von `DBIx::Class::Core` erben. Der Tabellenname, der der Methode `table` übergeben wird, darf nicht schon anderweitig verwendet werden. Wie bei anderen Klassen, die "normale" Tabellen repräsentieren, werden die Spalten definiert, die in der View existieren (mit `add_columns`).

Dem ResultSource-Objekt, das man mittels `result_source_instance` erhält, teilt man dann mit `name` mit, welche Abfrage hinter der View liegt. Dabei muss es sich um eine Referenz auf einen String handeln.

```
package DB::UserView;
use base qw/DBIx::Class::Core/;

my $stmt = qq~( SELECT username FROM
                usertabelle
                WHERE print_abo = 1 )~;

__PACKAGE__->table('my_user_view');
__PACKAGE__->add_columns(qw/username/);
__PACKAGE__->result_source_instance
           ->name(\$stmt);
```

Listing 4

In diesem Beispiel werden also alle User vorselektiert, die ein Print-Abonnement haben. In der Anwendung kann man sich in Zukunft die Bedingung sparen und stattdessen direkt auf die View zugreifen.

In der Schema-Klasse, die man auch vom Standard-Einsatz von `DBIx::Class` kennt, muss man die View-Klasse noch bekannt machen (Listing 5).

```
package MySchema;
use base qw/DBIx::Class::Schema/;

use DB::UserView;

__PACKAGE__->register_class(
  PrintAbo => 'DB::UserView'
);
```

Listing 5

Danach kann man in der Anwendung auf die View zugreifen wie auf jede andere Tabelle auch (siehe Listing 6).

```
#!/usr/bin/perl
use strict;
use warnings;

my $schema = MySchema->connect( ... );

my ($r) = $schema-> resultset('PrintAbo')
                 ->all;

print $r->username, "\n";
```

Listing 6

Die runden Klammern beim SQL-Befehl sind wichtig, da das ein Subselect ist. Betrachtet man das SQL-Statement, was bei der Verwendung abgesetzt wird, erkennt man warum das so ist:

```
SELECT me.username, me.print_abo FROM (
     SELECT username, print_abo
          FROM usertabelle )
     WHERE print_abo = 1 me
```

Listing 7

Ohne die Klammern würde es einen Syntax-Fehler geben.

Die Verwendung unterscheidet sich allerdings etwas, wenn in der View `?` verwendet werden. Wenn der Befehl also z.B. so aussieht:

```
my $stmt = qq~SELECT username
   FROM usertabelle
      WHERE age > ?~;
```

Listing 8

Dann muss bei der Abfrage in der Anwendung mit einem `bind`-Parameter gearbeitet werden:

```
my ($r) = $schema->resultset('PrintAbo')
     ->search( {}, {bind => [ 18 ]});
```

Listing 9

Profiling

Profiling wird immer dann betrieben, wenn die Anwendung zu langsam ist. Dabei geht es um die Suche der langsamsten Code-Stellen. Wenn man diese identifiziert hat, kann man sich hinsetzen und überlegen wie man etwas anders machen kann.

In diesem Abschnitt wird gezeigt, wie man SQL-Queries mit `DBIx::Class` profilen kann.

In `DBIx::Class` gibt es die Möglichkeit, SQL-Queries zu debuggen. Dazu muss das Debugging in `DBIx::Class::Storage` angeschaltet werden:

```
$dbic->storage->debug( 1 );
```

Das muss auch für das Profiling angeschaltet werden. Zusätzlich kann man eine Subklasse von `DBIx::Class::Storage::Statistics` erstellen, die eigene Profiling Mechanismen implementieren. Eine Beispielklasse ist in Listing 10 gezeigt.

```
package DB::Profiler;

use strict;
use warnings;
use DBIx::Class::Storage::Statistics;

our @ISA = qw(
         DBIx::Class::Storage::Statistics);
use Time::HiRes qw(time);

my $start;

sub query_start {
    $start = time;
}

sub query_end {
    my $diff = time - $start;
    my ($self,$sql,@params) = @_;

    print "Statement: $sql\n",
          "Parameters: @params\n",
          "Execution time: $diff\n\n";
    $start = undef;
}
```

Listing 10

In dem Beispiel werden die beiden Methoden `query_start` und `query_end` überschrieben. Wie der Name schon andeutet, werden die Methoden vor- bzw. nach dem Ausführen der Abfrage aufgerufen. Soll das Profiling auf Transaktionsebene stattfinden, so kann man die Methoden `txn_begin`, `txn_rollback` und `txn_commit` überschreiben.

Im Anwendungscode muss das Profiling noch aktiviert werden. Wie oben beschrieben muss zum einen der Debug-Modus eingeschaltet werden und zum anderen muss das Debugging-Objekt an das Storage-Objekt übergeben werden:

```
__PACKAGE__->storage
           ->debugobj(
               DB::Profiler->new
);
__PACKAGE__->storage->debug(1);
```

Listing 11

Sollen mit dem Profiling noch mehr Statistiken erstellt werden, so kann man das auf diese Weise machen:

```
sub query_end {
  my ($self,$sql,@params) = @_;

  my $elapsed = time - $start;
  push(@{ $calls{$sql} }, {
      params => \@params,
      elapsed => $elapsed
  });
}
```

Listing 12

Damit kann man dann für jeden SQL-Befehl statistische Daten wie Maximum-, Minimum- und Mittelwert berechnen und so feststellen, ob ein Befehl bei bestimmten Parametern besonders langsam ist. Eine weitere Empfehlung ist `DBIx::Class::QueryLog`.

Renée Bäcker