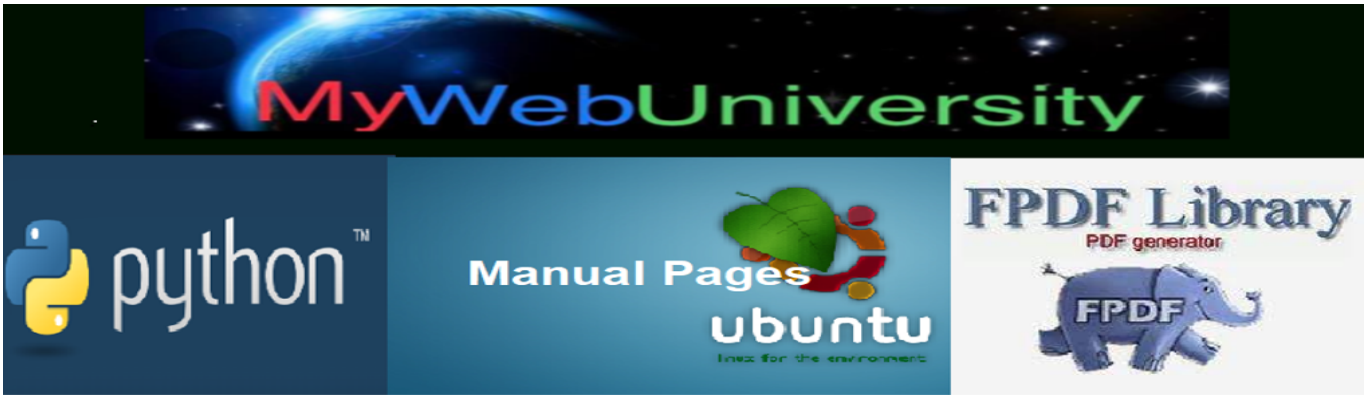




## *Rocky Enterprise Linux 9.2 Manual Pages on command 'DBI::DBD::SqlEngine::HowTo.3pm'*

***$ man DBI::DBD::SqlEngine::HowTo.3pm***

DBI::DBD::SqlEngine::HowTo(3pm)User Contributed Perl DocumentationDBI::DBD::SqlEngine::HowTo(3pm)

NAME

DBI::DBD::SqlEngine::HowTo - Guide to create DBI::DBD::SqlEngine based driver

SYNOPSIS

perldoc DBI::DBD::SqlEngine::HowTo

perldoc DBI

perldoc DBI::DBD

perldoc DBI::DBD::SqlEngine::Developers

perldoc SQL::Eval

perldoc DBI::DBD::SqlEngine

perldoc DBI::DBD::SqlEngine::HowTo

perldoc SQL::Statement::Embed

DESCRIPTION

This document provides a step-by-step guide, how to create a new "DBI::DBD::SqlEngine" based DBD. It expects that you carefully read the DBI documentation and that you're familiar with DBI::DBD and had read and understood DBD::ExampleP.

This document addresses experienced developers who are really sure that they need to invest time when writing a new DBI Driver. Writing a DBI Driver is neither a weekend project nor an easy job for hobby coders after work. Expect one or two man-month of time for the first start.

Those who are still reading, should be able to sing the rules of "CREATING A NEW DRIVER" in DBI::DBD.

CREATING DRIVER CLASSES

Do you have an entry in DBI's DBD registry? DBI::DBD::SqlEngine expect having a unique prefix for every driver class in inheritance chain.

It's easy to get a prefix - just drop the DBI team a note ("GETTING_HELP" in DBI). If you want for some reason hide your work, take a look at Class::Method::Modifiers how to wrap a private prefix method around existing "driver_prefix".

For this guide, a prefix of "foo_" is assumed.

## Sample Skeleton

```
package DBD::Foo;
use strict;
use warnings;
use vars qw($VERSION);
use base qw(DBI::DBD::SqlEngine);
use DBI ();
$VERSION = "0.001";
package DBD::Foo::dr;
use vars qw(@ISA $imp_data_size);
@ISA = qw(DBI::DBD::SqlEngine::dr);
$imp_data_size = 0;
package DBD::Foo::db;
use vars qw(@ISA $imp_data_size);
@ISA = qw(DBI::DBD::SqlEngine::db);
$imp_data_size = 0;
package DBD::Foo::st;
use vars qw(@ISA $imp_data_size);
@ISA = qw(DBI::DBD::SqlEngine::st);
$imp_data_size = 0;
package DBD::Foo::Statement;
use vars qw(@ISA);
@ISA = qw(DBI::DBD::SqlEngine::Statement);
package DBD::Foo::Table;
use vars qw(@ISA);
@ISA = qw(DBI::DBD::SqlEngine::Table);
1;
```

Tiny, eh? And all you have now is a DBD named foo which will is able to deal with temporary tables, as long as you use SQL::Statement. In DBI::SQL::Nano environments, this DBD can do nothing.

## Deal with own attributes

Before we start doing usable stuff with our DBI driver, we need to think about what we want to do and how we want to do it.

Do we need tunable knobs accessible by users? Do we need status information? All this is handled in attributes of the database handles (be careful when your DBD is running "behind" a DBD::Gofer proxy).

How come the attributes into the DBD and how are they fetchable by the user? Good question, but you should know because you've read the DBI documentation. "DBI::DBD::SqlEngine::db::FETCH" and "DBI::DBD::SqlEngine::db::STORE" taking care for you - all they need to know is which attribute names are valid and mutable or immutable. Tell them by adding "init_valid_attributes" to your db class:

```
sub init_valid_attributes
{
    my $dbh = $_[0];
    $dbh->SUPER::init_valid_attributes ();
    $dbh->{foo_valid_attrs} = {
        foo_version         => 1,   # contains version of this driver
        foo_valid_attrs     => 1,   # contains the valid attributes of foo drivers
        foo_readonly_attrs  => 1,   # contains immutable attributes of foo drivers
        foo_bar             => 1,   # contains the bar attribute
        foo_baz             => 1,   # contains the baz attribute
        foo_manager         => 1,   # contains the manager of the driver instance
        foo_manager_type    => 1,   # contains the manager class of the driver instance
    };
    $dbh->{foo_readonly_attrs} = {
        foo_version         => 1,   # ensure no-one modifies the driver version
        foo_valid_attrs     => 1,   # do not permit one to add more valid attributes ...
        foo_readonly_attrs  => 1,   # ... or make the immutable mutable
        foo_manager         => 1,   # manager is set internally only
    };
```

```
    return $dbh;
}
```

Woooho - but now the user cannot assign new managers? This is intended, overwrite "STORE" to handle it!

```
sub STORE ($$$)
{
    my ( $dbh, $attrib, $value ) = @_;
    $dbh->SUPER::STORE( $attrib, $value );
    # we're still alive, so no exception is thrown ...
    # by DBI::DBD::SqlEngine::db::STORE
    if ( $attrib eq "foo_manager_type" )
    {
        $dbh->{foo_manager} = $dbh->{foo_manager_type}->new();
        # ... probably correct some states based on the new
        # foo_manager_type - see DBD::Sys for an example
    }
}
```

But ... my driver runs without a manager until someone first assignes a "foo_manager_type". Well, no - there're two places where you can initialize defaults:

```
sub init_default_attributes
{
    my ($dbh, $phase) = @_;
    $dbh->SUPER::init_default_attributes($phase);
    if( 0 == $phase )
    {
        # init all attributes which have no knowledge about
        # user settings from DSN or the attribute hash
        $dbh->{foo_manager_type} = "DBD::Foo::Manager";
    }
    elsif( 1 == $phase )
    {
        # init phase with more knowledge from DSN or attribute
        # hash
```

```
            $dbh->{foo_manager} = $dbh->{foo_manager_type}->new();
        }
        return $dbh;
    }
```

So far we can prevent the users to use our database driver as data storage for anything and everything. We care only about the real important stuff for peace on earth and alike attributes. But in fact, the driver still can't do anything. It can do less than nothing - meanwhile it's not a stupid storage area anymore.

User comfort

"DBI::DBD::SqlEngine" since 0.05 consolidates all persistent meta data of a table into a single structure stored in "$dbh->{sql_meta}". While DBI::DBD::SqlEngine provides only readonly access to this structure, modifications are still allowed.

Primarily DBI::DBD::SqlEngine provides access via the setters "new_sql_engine_meta", "get_sql_engine_meta", "get_single_table_meta", "set_single_table_meta", "set_sql_engine_meta" and "clear_sql_engine_meta". Those methods are easily accessible by the users via the "$dbh->func ()" interface provided by DBI. Well, many users don't feel comfortize when calling

```
    # don't require extension for tables cars
    $dbh->func ("cars", "f_ext", ".csv", "set_sql_engine_meta");
```

DBI::DBD::SqlEngine will inject a method into your driver to increase the user comfort to allow:

```
    # don't require extension for tables cars
    $dbh->foo_set_meta ("cars", "f_ext", ".csv");
```

Better, but here and there users likes to do:

```
    # don't require extension for tables cars
    $dbh->{foo_tables}->{cars}->{f_ext} = ".csv";
```

This interface is provided when derived DBD's define following in "init_valid_attributes" (re-capture "Deal with own attributes"):

```
    sub init_valid_attributes
    {
        my $dbh = $_[0];
        $dbh->SUPER::init_valid_attributes ();
        $dbh->{foo_valid_attrs} = {
```

```
        foo_version         => 1,   # contains version of this driver
        foo_valid_attrs     => 1,   # contains the valid attributes of foo drivers
        foo_readonly_attrs  => 1,   # contains immutable attributes of foo drivers
        foo_bar             => 1,   # contains the bar attribute
        foo_baz             => 1,   # contains the baz attribute
        foo_manager         => 1,   # contains the manager of the driver instance
        foo_manager_type    => 1,   # contains the manager class of the driver instance
        foo_meta            => 1,   # contains the public interface to modify table meta attributes
      };
      $dbh->{foo_readonly_attrs} = {
        foo_version         => 1,   # ensure no-one modifies the driver version
        foo_valid_attrs     => 1,   # do not permit one to add more valid attributes ...
        foo_readonly_attrs  => 1,   # ... or make the immutable mutable
        foo_manager         => 1,   # manager is set internally only
        foo_meta            => 1,   # ensure public interface to modify table meta attributes are immutable
      };
      $dbh->{foo_meta} = "foo_tables";
      return $dbh;
    }
```

This provides a tied hash in "$dbh->{foo_tables}" and a tied hash for each table's meta data in "$dbh->{foo_tables}->{$table_name}". Modifications on the table meta attributes are done using the table methods:

```
    sub get_table_meta_attr { ... }
    sub set_table_meta_attr { ... }
```

Both methods can adjust the attribute name for compatibility reasons, e.g. when former versions of the DBD allowed different names to be used for the same flag:

```
    my %compat_map = (
                      abc => 'foo_abc',
                      xyz => 'foo_xyz',
                     );
    __PACKAGE__->register_compat_map( \%compat_map );
```

If any user modification on a meta attribute needs reinitialization of the meta structure (in case of "DBI::DBD::SqlEngine" these are the attributes "f_file", "f_dir", "f_ext" and

"f_lockfile"), inform DBI::DBD::SqlEngine by doing

```
my %reset_on_modify = (
                        foo_xyz => "foo_bar",
                        foo_abc => "foo_bar",
                      );
__PACKAGE__->register_reset_on_modify( \%reset_on_modify );
```

The next access to the table meta data will force DBI::DBD::SqlEngine to re-do the entire meta initialization process.

Any further action which needs to be taken can handled in "table_meta_attr_changed":

```
sub table_meta_attr_changed
{
    my ($class, $meta, $attrib, $value) = @_;
    ...
    $class->SUPER::table_meta_attr_changed ($meta, $attrib, $value);
}
```

This is done before the new value is set in $meta, so the attribute changed handler can act depending on the old value.

### Dealing with Tables

Let's put some life into it - it's going to be time for it.

This is a good point where a quick side step to SQL::Statement::Embed will help to shorten the next paragraph. The documentation in SQL::Statement::Embed regarding embedding in own DBD's works pretty fine with SQL::Statement and DBI::SQL::Nano.

Second look should go to DBI::DBD::SqlEngine::Developers to get a picture over the driver part of the table API. Usually there isn't much to do for an easy driver.

### Testing

Now you should have your first own DBD. Was easy, wasn't it?  But does it work well? Prove it by writing tests and remember to use dbd_edit_mm_attribs from DBI::DBD to ensure testing even rare cases.

## AUTHOR

This guide is written by Jens Rehsack. DBI::DBD::SqlEngine is written by Jens Rehsack using code from DBD::File originally written by Jochen Wiedmann and Jeff Zucker.

The module DBI::DBD::SqlEngine is currently maintained by

H.Merijn Brand < h.m.brand at xs4all.nl > and Jens Rehsack  < rehsack at googlemail.com >

## COPYRIGHT AND LICENSE