NAME

DBI::DBD::SqlEngine::Developers - Developers documentation for DBI::DBD::SqlEngine

SYNOPSIS

```
package DBD::myDriver;

use base qw(DBI::DBD::SqlEngine);

sub driver
{
...
my $drh = $proto->SUPER::driver($attr);
...
return $drh->{class};
}

sub CLONE { ... }

package DBD::myDriver::dr;

@ISA = qw(DBI::DBD::SqlEngine::dr);

sub data_sources { ... }
...

package DBD::myDriver::db;

@ISA = qw(DBI::DBD::SqlEngine::db);

sub init_valid_attributes { ... }
sub init_default_attributes { ... }
sub set_versions { ... }
sub validate_STORE_attr { my ($dbh, $attrib, $value) = @_; ... }
sub validate_FETCH_attr { my ($dbh, $attrib) = @_; ... }
sub get_myd_versions { ... }
sub get_avail_tables { ... }

package DBD::myDriver::st;

@ISA = qw(DBI::DBD::SqlEngine::st);

sub FETCH { ... }
sub STORE { ... }

package DBD::myDriver::Statement;

@ISA = qw(DBI::DBD::SqlEngine::Statement);

sub open_table { ... }

package DBD::myDriver::Table;

@ISA = qw(DBI::DBD::SqlEngine::Table);

my %reset_on_modify = (
```

```
myd_abc => "myd_foo",
myd_mno => "myd_bar",
);
__PACKAGE__->register_reset_on_modify( \%reset_on_modify );
my %compat_map = (
abc => 'foo_abc',
xyz => 'foo_xyz',
);
__PACKAGE__->register_compat_map( \%compat_map );

sub bootstrap_table_meta { ... }
sub init_table_meta { ... }
sub table_meta_attr_changed { ... }
sub open_data { ... }

sub new { ... }

sub fetch_row { ... }
sub push_row { ... }
sub push_names { ... }
sub seek { ... }
sub truncate { ... }
sub drop { ... }

# optimize the SQL engine by add one or more of
sub update_current_row { ... }
# or
sub update_specific_row { ... }
# or
sub update_one_row { ... }
# or
sub insert_new_row { ... }
# or
sub delete_current_row { ... }
# or
sub delete_one_row { ... }
```

DESCRIPTION

This document describes the interface of DBI::DBD::SqlEngine for DBD developers who write DBI::DBD::SqlEngine based DBI drivers. It supplements DBI::DBD and DBI::DBD::SqlEngine::HowTo, which you should read first.

CLASSES

Each DBI driver must provide a package global `driver` method and three DBI related classes:

DBI::DBD::SqlEngine::dr

Driver package, contains the methods DBI calls indirectly via DBI interface:

```
DBI->connect ('DBI:DBM:', undef, undef, {})

# invokes
package DBD::DBM::dr;
@DBD::DBM::dr::ISA = qw(DBI::DBD::SqlEngine::dr);

sub connect ($$;$$)
{
...
```

```
}
```

Similar for `data_sources ()` and `disconnect_all()`.

Pure Perl DBI drivers derived from DBI::DBD::SqlEngine usually don't need to override any of the methods provided through the DBD::XXX::dr package. However if you need additional initialization not fitting in `init_valid_attributes()` and `init_default_attributes()` of you're ::db class, the connect method might be the final place to be modified.

DBI::DBD::SqlEngine::db
: Contains the methods which are called through DBI database handles (`$dbh`). e.g.,

```
$sth = $dbh->prepare ("select * from foo");
# returns the f_encoding setting for table foo
$dbh->csv_get_meta ("foo", "f_encoding");
```

DBI::DBD::SqlEngine provides the typical methods required here. Developers who write DBI drivers based on DBI::DBD::SqlEngine need to override the methods `set_versions` and `init_valid_attributes`.

DBI::DBD::SqlEngine::TieMeta;
: Provides the tie-magic for `$dbh->{$drv_pfx . "_meta"}`. Routes `STORE` through `$drv->set_sql_engine_meta()` and `FETCH` through `$drv->get_sql_engine_meta()`. `DELETE` is not supported, you have to execute a `DROP TABLE` statement, where applicable.

DBI::DBD::SqlEngine::TieTables;
: Provides the tie-magic for tables in `$dbh->{$drv_pfx . "_meta"}`. Routes`STORE` though `$tblClass->set_table_meta_attr()` and `FETCH` though `$tblClass->get_table_meta_attr()`. `DELETE` removes an attribute from the *meta object* retrieved by `$tblClass->get_table_meta()`.

DBI::DBD::SqlEngine::st
: Contains the methods to deal with prepared statement handles. e.g.,

```
$sth->execute () or die $sth->errstr;
```

DBI::DBD::SqlEngine::TableSource;
: Base class for 3rd party table sources:

```
$dbh->{sql_table_source} = "DBD::Foo::TableSource";
```

DBI::DBD::SqlEngine::DataSource;
: Base class for 3rd party data sources:

```
$dbh->{sql_data_source} = "DBD::Foo::DataSource";
```

DBI::DBD::SqlEngine::Statement;
: Base class for derived drivers statement engine. Implements `open_table`.

DBI::DBD::SqlEngine::Table;
: Contains tailoring between SQL engine's requirements and `DBI::DBD::SqlEngine` magic for finding the right tables and storage. Builds bridges between `sql_meta` handling of `DBI::DBD::SqlEngine::db` table initialization for SQL engines and *meta object*'s attribute management for derived drivers.

DBI::DBD::SqlEngine
: This is the main package containing the routines to initialize DBI::DBD::SqlEngine based DBI drivers. Primarily the `DBI::DBD::SqlEngine::driver` method is invoked, either directly from DBI when the driver is initialized or from the derived class.

```
package DBD::DBM;

use base qw( DBI::DBD::SqlEngine );
```

```
sub driver
{
my ( $class, $attr ) = @_;
...
my $drh = $class->SUPER::driver( $attr );
...
return $drh;
}
```

It is not necessary to implement your own driver method as long as additional initialization (e.g. installing more private driver methods) is not required. You do not need to call `setup_driver` as DBI::DBD::SqlEngine takes care of it.

DBI::DBD::SqlEngine::dr

The driver package contains the methods DBI calls indirectly via the DBI interface (see "DBI Class Methods" in DBI).

DBI::DBD::SqlEngine based DBI drivers usually do not need to implement anything here, it is enough to do the basic initialization:

```
package DBD:XXX::dr;

@DBD::XXX::dr::ISA = qw (DBI::DBD::SqlEngine::dr);
$DBD::XXX::dr::imp_data_size = 0;
$DBD::XXX::dr::data_sources_attr = undef;
$DBD::XXX::ATTRIBUTION = "DBD::XXX $DBD::XXX::VERSION by Hans Mustermann";
```

Methods provided by `DBI::DBD::SqlEngine::dr`

connect

Supervises the driver bootstrap when calling

```
DBI->connect( "dbi:Foo", , , { ... } );
```

First it instantiates a new driver using `DBI::_new_dbh` After that, initial bootstrap of the newly instantiated driver is done by

```
$dbh->func( 0, "init_default_attributes" );
```

The first argument (`0`) signals that this is the very first call to `init_default_attributes`. Modern drivers understand that and do early stage setup here after calling

```
package DBD::Foo::db;
our @DBD::Foo::db::ISA = qw(DBI::DBD::SqlEngine::db);

sub init_default_attributes
{
my ($dbh, $phase) = @_;
$dbh->SUPER::init_default_attributes($phase);
...; # own setup code, maybe separated by phases
}
```

When the `$phase` argument is passed down until `DBI::DBD::SqlEngine::db::init_default_attributes` `connect()` recognizes a *modern* driver and initializes the attributes from *DSN* and `attr` arguments passed via `DBI->connect( $dsn, $user, $pass, \%attr )`.

At the end of the attribute initialization after *phase 0*, `connect()` invoked `init_default_attributes` again for *phase 1*:

```
$dbh->func( 1, "init_default_attributes" );
```

data_sources
: Returns a list of *DSN*'s using the `data_sources` method of the class specified in `$dbh->{sql_table_source}` or via `\%attr`:

```
@ary = DBI->data_sources($driver);
@ary = DBI->data_sources($driver, \%attr);
```

disconnect_all
: DBI::DBD::SqlEngine doesn't have an overall driver cache, so nothing happens here at all.

DBI::DBD::SqlEngine::db

This package defines the database methods, which are called via the DBI database handle `$dbh`.

Methods provided by `DBI::DBD::SqlEngine::db`

ping
: Simply returns the content of the `Active` attribute. Override when your driver needs more complicated actions here.

prepare
: Prepares a new SQL statement to execute. Returns a statement handle, `$sth` - instance of the DBD:XXX::st. It is neither required nor recommended to override this method.

validate_FETCH_attr
: Called by `FETCH` to allow inherited drivers do their own attribute name validation. Calling convention is similar to `FETCH` and the return value is the approved attribute name.

```
return $validated_attribute_name;
```

In case of validation fails (e.g. accessing private attribute or similar), `validate_FETCH_attr` is permitted to throw an exception.

FETCH
: Fetches an attribute of a DBI database object. Private handle attributes must have a prefix (this is mandatory). If a requested attribute is detected as a private attribute without a valid prefix, the driver prefix (written as `$drv_prefix`) is added.

The driver prefix is extracted from the attribute name and verified against `$dbh->{ $drv_prefix . "valid_attrs" }`(when it exists). If the requested attribute v alue is not listed as a valid attribute, this method croaks. If the attribute is valid and readonly (listed in `$dbh->{ $drv_prefix . "readonly_attrs" }`when it exists), a real cop y of the attribute value is returned. So it's not possible to modify `f_valid_attrs` from outside of DBI::DBD::SqlEngine::db or a derived class.

validate_STORE_attr
: Called by `STORE` to allow inherited drivers do their own attribute name validation. Calling convention is similar to `STORE` and the return value is the approved attribute name followed by the approved new value.

```
return ($validated_attribute_name, $validated_attribute_value);
```

In case of validation fails (e.g. accessing private attribute or similar), `validate_STORE_attr` is permitted to throw an exception (`DBI::DBD::SqlEngine::db::validate_STORE_attr` throws an exception when someone tries to assign value other than `SQL_IC_UPPER .. SQL_IC_MIXED` to `$dbh->{sql_identifier_case}` or `$dbh->{sql_quoted_identifier_case}`).

STORE
: Stores a database private attribute. Private handle attributes must have a prefix (this is mandatory). If a requested attribute is detected as a private attribute without a valid prefix, the driver prefix (written as `$drv_prefix`) is added. If the database handle has an attribute

`${drv_prefix}_valid_attrs` - for attribute names which are not listed in that hash, this method croaks. If the database handle has an attribute `${drv_prefix}_readonly_attrs`, only attributes which are not listed there can be stored (once they are initialized). Trying to overwrite such an immutable attribute forces this method to croak.

An example of a valid attributes list can be found in `DBI::DBD::SqlEngine::db::init_valid_attributes`

set_versions
This method sets the attributes `f_version`, `sql_nano_version`, `sql_statement_version` and (if not prohibited by a restrictive `${prefix}_valid_attrs`) `${prefix}_version`.

This method is called at the end of the `connect ()` phase.

When overriding this method, do not forget to invoke the superior one.

init_valid_attributes
This method is called after the database handle is instantiated as the first attribute initialization.

`DBI::DBD::SqlEngine::db::init_valid_attributes` initializes the attributes `sql_valid_attrs` and `sql_readonly_attrs`.

When overriding this method, do not forget to invoke the superior one, preferably before doing anything else.

init_default_attributes
This method is called after the database handle is instantiated to initialize the default attributes. It expects one argument: `$phase`. If`$phase` is not giv en, `connect` of `DBI::DBD::SqlEngine::dr` expects this is an old-fashioned driver which isn't capable of multi-phased initialization.

`DBI::DBD::SqlEngine::db::init_default_attributes` initializes the attributes `sql_identifier_case`, `sql_quoted_identifier_case`, `sql_handler`, `sql_init_order`, `sql_meta`, `sql_engine_version`, `sql_nano_version` and `sql_statement_version` when SQL::Statement is available.

It sets `sql_init_order` to the given `$phase`.

When the derived implementor class provides the attribute to validate attributes (e.g. `$dbh->{dbm_valid_attrs} = {...};`) or the attribute containing the immutable attributes (e.g. `$dbh->{dbm_readonly_attrs} = {...};`), the attributes `drv_valid_attrs`, `drv_readonly_attrs` and `drv_version` are added (when available) to the list of valid and immutable attributes (where `drv_` is interpreted as the driver prefix).

get_versions
This method is called by the code injected into the instantiated driver to provide the user callable driver method `${prefix}versions` (e.g. `dbm_versions`, `csv_versions`, ...).

The DBI::DBD::SqlEngine implementation returns all version information known by DBI::DBD::SqlEngine (e.g. DBI version, Perl version, DBI::DBD::SqlEngine version and the SQL handler version).

`get_versions` takes the `$dbh` as the first argument and optionally a second argument containing a table name. The second argument is not evaluated in `DBI::DBD::SqlEngine::db::get_versions` itself - but might be in the future.

If the derived implementor class provides a method named `get_${drv_prefix}versions`, this is invoked and the return value of it is associated to the derived driver name:

```
if (my $dgv = $dbh->{ImplementorClass}->can ("get_" . $drv_prefix . "versions") {
(my $derived_driver = $dbh->{ImplementorClass}) = s/::db$//;
$versions{$derived_driver} = &$dgv ($dbh, $table);
}
```

Override it to add more version information about your module, (e.g. some kind of parser version in case of DBD::CSV, ...), if one line is not enough room to provide all relevant information.

sql_parser_object
: Returns a SQL::Parser instance, when `sql_handler` is set to "SQL::Statement" The parser instance is stored in `sql_parser_object`.

  It is not recommended to override this method.

disconnect
: Disconnects from a database. All local table information is discarded and the `Active` attribute is set to 0.

type_info_all
: Returns information about all the types supported by DBI::DBD::SqlEngine.

table_info
: Returns a statement handle which is prepared to deliver information about all known tables.

list_tables
: Returns a list of all known table names.

quote
: Quotes a string for use in SQL statements.

commit
: Warns about a useless call (if warnings enabled) and returns. DBI::DBD::SqlEngine is typically a driver which commits every action instantly when executed.

rollback
: Warns about a useless call (if warnings enabled) and returns. DBI::DBD::SqlEngine is typically a driver which commits every action instantly when executed.

Attributes used by `DBI::DBD::SqlEngine::db`

This section describes attributes which are important to developers of DBI Database Drivers derived from `DBI::DBD::SqlEngine`

sql_init_order
: This attribute contains a hash with priorities as key and an array containing the `$dbh` attributes to be initialized during before/after other attributes.

  `DBI::DBD::SqlEngine` initializes following attributes:

```
$dbh->{sql_init_order} = {
0 => [qw( Profile RaiseError PrintError AutoCommit )],
90 => [ "sql_meta", $dbh->{$drv_pfx_meta} ? $dbh->{$drv_pfx_meta} : () ]
}
```

  The default priority of not listed attribute keys is `50`. It is well known that a lot of attributes needed to be set before some table settings are initialized. For example, for DBD::DBM when using

```
my $dbh = DBI->connect( "dbi:DBM:", undef, undef, {
f_dir => "/path/to/dbm/databases",
dbm_type => "BerkeleyDB",
dbm_mldbm => "JSON", # use MLDBM::Serializer::JSON
dbm_tables => {
quick => {
dbm_type => "GDBM_File",
dbm_MLDBM => "FreezeThaw"
}
}
});
```

This defines a known table `quick` which uses the GDBM_File backend and FreezeThaw as serializer instead of the overall default BerkeleyDB and JSON. **But** all files containing the table data have to be searched in `$dbh->{f_dir}`, which requires `$dbh->{f_dir}` must be initialized before `$dbh->{sql_meta}->{quick}` is initialized by `bootstrap_table_meta` method of ''DBI::DBD::SqlEngine::Table'' to get `$dbh->{sql_meta}->{quick}->{f_dir}` being initialized properly.

sql_init_phase
: This attribute is only set during the initialization steps of the DBI Database Driver. It contains the value of the currently run initialization phase. Currently supported phases are *phase 0*and *phase 1*. This attribute is set in `init_default_attributes` and remo ved in `init_done`.

sql_engine_in_gofer
: This value has a true value in case of this driver is operated via DBD::Gofer. The impact of being operated via Gofer is a read-only driver (not read-only databases!), so you cannot modify any attributes later - neither any table settings. **But** you won't get an error in cases you modify table attributes, so please carefully watch `sql_engine_in_gofer`.

sql_table_source
: Names a class which is responsible for delivering *data sources* and *available tables* (Database Driver related). *data sources* here refers to ''data_sources'' in DBI, not `sql_data_source`.

  See ''DBI::DBD::SqlEngine::TableSource'' for details.

sql_data_source
: Name a class which is responsible for handling table resources open and completing table names requested via SQL statements.

  See ''DBI::DBD::SqlEngine::DataSource'' for details.

sql_dialect
: Controls the dialect understood by SQL::Parser. Possible values (delivery state of SQL::Statement):

  ```
  * ANSI
  * CSV
  * AnyData
  ```

  Defaults to ''CSV''. Because an SQL::Parser is instantiated only once and SQL::Parser doesn't allow to modify the dialect once instantiated, it's strongly recommended to set this flag before any statement is executed (best place is connect attribute hash).

DBI::DBD::SqlEngine::st
: Contains the methods to deal with prepared statement handles:

  bind_param
  : Common routine to bind placeholders to a statement for execution. It is dangerous to override this method without detailed knowledge about the DBI::DBD::SqlEngine internal

storage structure.

execute
: Executes a previously prepared statement (with placeholders, if any).

finish
: Finishes a statement handle, discards all buffered results. The prepared statement is not discarded so the statement can be executed again.

fetch
: Fetches the next row from the result-set. This method may be rewritten in a later version and if it's overridden in a derived class, the derived implementation should not rely on the storage details.

fetchrow_arrayref
: Alias for `fetch`.

FETCH
: Fetches statement handle attributes. Supported attributes (for full overview see "Statement Handle Attributes" in DBI) are `NAME`, `TYPE`, `PRECISION` and `NULLABLE`. Each column is returned as `NULLABLE` which might be wrong depending on the derived backend storage. If the statement handle has private attributes, they can be fetched using this method, too. **Note** that statement attributes are not associated with any table used in this statement.

  This method usually requires extending in a derived implementation. See DBD::CSV or DBD::DBM for some example.

STORE
: Allows storing of statement private attributes. No special handling is currently implemented here.

rows
: Returns the number of rows affected by the last execute. This method might return `undef`.

DBI::DBD::SqlEngine::TableSource

Provides data sources and table information on database driver and database handle level.

```
package DBI::DBD::SqlEngine::TableSource;

sub data_sources ($;$)
{
my ( $class, $drh, $attrs ) = @_;
...
}

sub avail_tables
{
my ( $class, $drh ) = @_;
...
}
```

The `data_sources` method is called when the user invokes any of the following:

```
@ary = DBI->data_sources($driver);
@ary = DBI->data_sources($driver, \%attr);

@ary = $dbh->data_sources();
@ary = $dbh->data_sources(\%attr);
```

The `avail_tables` method is called when the user invokes any of the following:

```
@names = $dbh->tables( $catalog, $schema, $table, $type );
```

```
$sth = $dbh->table_info( $catalog, $schema, $table, $type );
$sth = $dbh->table_info( $catalog, $schema, $table, $type, \%attr );

$dbh->func( "list_tables" );
```

Every time where an `\%attr` argument can be specified, this `\%attr` object's `sql_table_source` attribute is preferred over the `$dbh` attribute or the driver default.

DBI::DBD::SqlEngine::DataSource

Provides base functionality for dealing with tables. It is primarily designed for allowing transparent access to files on disk or already opened (file-)streams (e.g. for DBD::CSV

Derived classes shall be restricted to similar functionality, too (e.g. opening streams from an archive, transparently compress/uncompress log files before parsing them,

```
package DBI::DBD::SqlEngine::DataSource;

sub complete_table_name ($$;$)
{
my ( $self, $meta, $table, $respect_case ) = @_;
...
}
```

The method `complete_table_name` is called when first setting up the *meta information* for a table:

```
"SELECT user.id, user.name, user.shell FROM user WHERE ..."
```

results in opening the table `user`. First step of the table open process is completing the name. Let's imagine you're having a DBD::CSV handle with following settings:

```
$dbh->{sql_identifier_case} = SQL_IC_LOWER;
$dbh->{f_ext} = '.lst';
$dbh->{f_dir} = '/data/web/adrmgr';
```

Those settings will result in looking for files matching `[Uu][Ss][Ee][Rr](\.lst)?$` in `/data/web/adrmgr/`. The scanning of the directory `/data/web/adrmgr/` and the pattern match check will be done in `DBD::File::DataSource::File` by the `complete_table_name` method.

If you intend to provide other sources of data streams than files, in addition to provide an appropriate `complete_table_name` method, a method to open the resource is required:

```
package DBI::DBD::SqlEngine::DataSource;

sub open_data ($)
{
my ( $self, $meta, $attrs, $flags ) = @_;
...
}
```

After the method `open_data` has been run successfully, the table's meta information are in a state which allows the table's data accessor methods will be able to fetch/store row information. Implementation details heavily depends on the table implementation, whereby the most famous is surely DBD::File::Table.

DBI::DBD::SqlEngine::Statement

Derives from DBI::SQL::Nano::Statement for unified naming when deriving new drivers. No additional feature is provided from here.

DBI::DBD::SqlEngine::Table

Derives from DBI::SQL::Nano::Table for unified naming when deriving new drivers.

You should consult the documentation of `SQL::Eval::Table` (see SQL::Eval) to get more

information about the abstract methods of the table's base class you have to override and a description of the table meta information expected by the SQL engines.

bootstrap_table_meta
: Initializes a table meta structure. Can be safely overridden in a derived class, as long as the `SUPER` method is called at the end of the overridden method.

  It copies the following attributes from the database into the table meta data `$dbh->{ReadOnly}` into `$meta->{readonly}`, `sql_identifier_case` and `sql_data_source` and makes them sticky to the table.

  This method should be called before you attempt to map between file name and table name to ensure the correct directory, extension etc. are used.

init_table_meta
: Initializes more attributes of the table meta data - usually more expensive ones (e.g. those which require class instantiations) - when the file name and the table name could mapped.

get_table_meta
: Returns the table meta data. If there are none for the required table, a new one is initialized. When after bootstrapping a new *table_meta* and completing the table name a mapping can be established between an existing *table_meta* and the new bootstrapped one, the already existing is used and a mapping shortcut between the recent used table name and the already known table name is hold in `$dbh->{sql_meta_map}`. When it fails, nothing is returned. On success, the name of the table and the meta data structure is returned.

get_table_meta_attr
: Returns a single attribute from the table meta data. If the attribute name appears in `%compat_map`, the attribute name is updated from there.

set_table_meta_attr
: Sets a single attribute in the table meta data. If the attribute name appears in `%compat_map`, the attribute name is updated from there.

table_meta_attr_changed
: Called when an attribute of the meta data is modified.

  If the modified attribute requires to reset a calculated attribute, the calculated attribute is reset (deleted from meta data structure) and the *initialized* flag is removed, too. The decision is made based on `%register_reset_on_modify`.

register_reset_on_modify
: Allows `set_table_meta_attr` to reset meta attributes when special attributes are modified. For DBD::File, modifying one of `f_file`, `f_dir`, `f_ext` or `f_lockfile` will reset `f_fqfn`. DBD::DBM extends the list for `dbm_type` and `dbm_mldbm` to reset the value of `dbm_tietype`.

  If your DBD has calculated values in the meta data area, then call `register_reset_on_modify`:

  ```
  my %reset_on_modify = ( "xxx_foo" => "xxx_bar" );
  __PACKAGE__->register_reset_on_modify( \%reset_on_modify );
  ```

register_compat_map
: Allows `get_table_meta_attr` and `set_table_meta_attr` to update the attribute name to the current favored one:

  ```
  # from DBD::DBM
  my %compat_map = ( "dbm_ext" => "f_ext" );
  __PACKAGE__->register_compat_map( \%compat_map );
  ```

open_data
: Called to open the table's data storage. This is silently forwarded to `$meta->{sql_data_source}->open_data()`.

After this is done, a derived class might add more steps in an overridden `open_file` method.

new

Instantiates the table. This is done in 3 steps:

1. `get the table meta data`
2. `open the data file`
3. `bless the table data structure using inherited constructor new`

It is not recommended to override the constructor of the table class. Find a reasonable place to add you extensions in one of the above four methods.

AUTHOR

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