MySQL and Microsoft SQL Server to Code

The real magic of QuickDRY is the ORM. QuickDRY is designed to work with legacy data sources and so it doesn’t ask you anything about it once you set up your basic parameters.

It will find:

* Primary Key
* Unique Keys
* Foreign Keys
* Stored Procedures

It will generate:

* Table Classes
* Stored Procedure Classes
* CRUD Dialog
  + Add/Edit
* CRUD API
  + Get
  + Delete
  + History
  + Lookup
  + Save
* Simple Table Management Page

All the code that finds the database scheme and interprets it is found in

QuickDRY/connectors/mssql/MSSQL\_CodeGen.php

The script to generate the code from your data warehouse is in /scripts. All it does is take your command line parameters and instantiate the code generation class using them.

# Table Classes

Some projects have a single database, some projects have multiple databases. Some projects have multiple database servers. Some data warehouses have unique names for every table across all data sources, some have duplicates. Depending on the complexity of your data warehouse you can configure code generation to ensure there are no PHP classes with duplicate names generated by the ORM.

There are two classes generated for each table and each set of tables is put into a folder based on the name of the database.

## Database Prefix

All SQL Server classes and files start with “ms\_” which prevents name conflicts between data warehouse sources and indicates the type of server these classes are talking to.

The second part of the name is either the actual database name or a constant you have defined to hold the name of the database.

For larger warehouses, the name of the database will not change between production and development environments. In those cases, you can simply use the name of the database.

For smaller warehouses, there is maybe only one database and the name of it depends on the environment. For example the database “transactions” may be “transactions\_dev” or “transactions\_20180101” depending on whether you are operating on development or historical data. But we obviously don’t want to create different table classes for each environment. So instead a constant is used:

define(‘TRANSACTIONS’,’transactions’);

The value of the define can be modified per environment using the settings.<host>.php files.

When using a constant the full database prefix will be

“ms\_transactions\_”

And inside each db file the database referenced will be the constant TRANSACTIONS

If you did not use a constant then the database referenced will be the string “transactions” and the classes will not work if you change the database name in the database without updating every reference in the generated code.

Whether you use a constant or the database name directly is set using the -c flag

-dtransactions -cTRANSACTIONS

Will create classes for all the tables in the transactions database but reference the TRANSACTIONS constant in the generated code.

-dtransactions

Will create classes for all the tables in the transactions database and use the string “transactions” in the generated code

## User Class

-u = the class name of your user objects (in QuickDRY this is by default “UserClass” but you may need to customize this)

-v = the variable name of the user (in QuickDRY this is by default “CurrentUser”)

-I = the property of the user class that is the primary key (in QuickDRY this is by default “id”)

This is needed to lock down the API endpoints by default so only users with the role defined by ROLE\_ID\_ADMIN can operate on database records through the API.

**public function** VisibleTo(UserClass &$user)  
{  
 **if**($user->Is([***ROLE\_ID\_ADMIN***])) {  
 **return true**;  
 }  
  
 **return false**;  
}  
  
**public function** CanDelete(UserClass &$user)  
{  
 **if**($user->Is([***ROLE\_ID\_ADMIN***])) {  
 **return true**;  
 }  
  
 **return false**;  
}

These methods are generated in the db file for each table and if needed, are overridden. If your website is more form driven then the default is fine because your pages will handle security. If your website is api driven then these are here to avoid inadvertent security holes.

# Db Files

These files are generated from the database tables and will be updated when the table changes.

NOTE: DO NOT EDIT THESE FILES

They are stored in the format

db/<database>/db\_ms\_<database><table>Class.php

After you run the code generation, you will copy and overwrite the existing files in httpdocs/common with the contents of db/

These files contain the classes which allow you to operate on the tables using PHP. They have the column definitions, unique key definitions, primary key and foreign key links.

If you have a foreign key that links A to B then by instantiating A as $A you will be able to access B using $A->B

If you instantiate B as $B you will have an array of A using foreach($B->A as $item). In addition there a Count property so you can get the count of A in $B. There is an IsReferenced() method that by default returns 0 if there are no foreign keys or returns the sum of the foreign key references. If B is referenced twice in the A table then it will return 2.

The Delete API file will check IsReferenced and return an error to the user if there are foreign key references to the row being deleted.

# Class Files

These files are initially generated from the database tables but mostly just extend the db files. These files are to be changed by the developer.

NOTE: Copy but do not overwrite files from scripts/class to httpdocs/common

These files are used to override the default behaviors of the DB files as well as implement business logic.

## Master Pages

The -m flag sets the master page constant that will be used in the basic management pages.

All master pages go in httpdocs/masterpages

If the filename is “default.php” then the define should be

Define(‘MASTERPAGE\_DEFAULT’,’default’);

You would then pass in -m”MASTERPAGE\_DEFAULT”

## Lowercase Table

QuickDRY tries to create camel case class names and being successful at that requires knowing the naming convention of the database.

If you use my\_table then lowercase can be set to zero or one. QuickDRY will explode on the \_ and upper case the letters before putting it back together giving you MyTable for the class name.

If you use MyTable in your database already then you will want to set lowercase to zero. If you set it to one, it will lowercase the name to mytable and then uppercase the words which gives you Mytable. Setting it to zero will preserve MyTable.

You may also have tables names such as My\_FBI\_Files in which case you’ll want to set it to zero and it will be changed to MyFBIFiles. If you set it to one then it will be changed to MyFbiFiles.

This is all purely cosmetic as the actual table name is referenced in the class. This only determines how the class will be named that you will use in your code.

Anything that is not a letter or number is changed to a space before ucwords is used which sets the first letter of each “word” to uppercase.

If a table name starts with a number then “i” is placed in front of the class name because PHP does not support class names that start with a number.

The logic that turns tables names into class names is contained in SQL\_Base:: TableToClass

## Use FK Column Name

This flag is useful for when you have a table referencing another table multiple times. If you were to only use the foreign table you would end up with two identical properties in the class which would cause your code to not compile.

Bank ->Customer => User

Bank->Manager => User

Would end up with User twice

When you set Use FK Column Name to one then the above would generate

Bank->User\_Manager

Bank->User\_Customer

User is first in the property name so you quickly find all your references to the User table and then the column in the current table is used to indicate which reference you want.

## Database Class

QuickDRY comes with three connection classes for MySQL and three connection classes for SQL Server. The code generation class defaults to MYSQL\_A and MSSQL\_A for MySQL and SQL Server. You can pass in any class and the db classes will extend the specified connection.

This database classes allow you to reach into data sources across multiple servers. Some businesses may have more than one physical database server. You can talk to both or 100 of them in your project by creating simple database classes to accommodate more unique connections.

<?php

/\*\*

\* Class MSSQL\_Base

\*/

class MSSQL\_A extends MSSQL\_Core

{

protected static $connection = null;

protected static function \_connect()

{

if(is\_null(static::$connection)) {

static::$DB\_HOST = MSSQL\_HOST;

static::$connection = new MSSQL\_Connection(MSSQL\_HOST, MSSQL\_USER, MSSQL\_PASS);

}

}

}

These classes simply allow you to specify more connection constants in your settings files. Nothing else goes in these files.

The included classes are in QuickDRY\connectors\mssql and QuickDRY\connectors\mysql however, your additional classes could go anywhere. They must be included before you can generate classes using them because they are used by the code generation to connect to your data source and get the schema information.

# Stored Procedures

While QuickDRY can figure out what stored procedures are in a database and figure out what the parameter names are, it cannot figure out what the data structure is that is returned.

That would require analyzing SQL or throwing random garbage into the parameters to see what comes back. Both of which are terrible ideas.

Instead QuickDRY can be configured to tell you what the resulting column set is the first time you try to use a stored procedure class.

**<?php***/\*\*  
 \* Class ms\_<database>\_<storedProcName>Class  
 \*/***class** ms\_<database>\_<storedProcName>Class **extends** SafeClass  
{  
 **public function** \_\_construct($row = **null**)  
 {  
 **if**($row) {  
 $this->HaltOnError(**false**);  
 $this->FromRow($row);  
 **if**($this->HasMissingProperties()) {  
 Halt($this->GetMissingPropeties());  
 }  
 }  
 }  
}

The default behavior of SafeClass is to halt whenever you try to get or set a property not explicitly defined in the class. This behavior can be overridden using the HaltOnError method. This changes the behavior to keep a record of all the properties referenced that are not define. You can then check to see if a list was created and then Halt on that list which generates PHP code you can copy and paste into the class definition to define those properties.

In the scripts folder there is a \_common folder. The reason for the underscore is so that PHP does not confuse it for the httpdocs/common when running the ORM.

In that folder you will find “sp\_” files. These should be copied to the httpdocs/common folder and overwrite whatever is already there.

Scripts/includes should be copied to httpdocs and overwrite whatever is these.

Scripts/includes files include the common/sp\_ files as well as all the other class definitions for all databases and tables.

The contents of scripts/sp should be copied to httpdocs/common but not overwrite what is already there. Otherwise you will lose your work.

Because Stored Procedure files are so small, they are not broken up into two files. They exist solely to allow you to run stored procedures and return an array of results.

You will use sp\_ms\_<database>::<stored proc name>($param…) to execute your stored procedure and pass in any parameters.

It will return an array of the type for that stored procedure.

Any further business logic will be done outside of the generated stored proc code.