Understanding Migrations and Schema Update with Doctrine in Mautic

If you have ever had to troubleshoot a failed Mautic upgrade, you might have come across this page, which, among other things, describes the commands **doctrine:migrations:migrate** and **doctrine:schema:update**.

If you have ever wondered what those two commands actually do, this is a short explanation. Hopefully, you will never have to use either of them, as the automated Mautic upgrade script takes care of everything, but on the off-chance that something goes wrong it can't hurt to know a bit about them.

Neither of the two commands are directly related to Mautic - rather, they are related to the PHP framework that Mautic is built on, Symfony.

The *doctrine:migrations* commands are handled by the DoctrineMigrationsBundle and are essentially a safer way of updating the database compared to the *doctrine:schema:update*. As for what *doctrine:schema:update* is, we need to go back a bit and look at how Mautic (powered by Symfony) handles the MySQL database:

Object Relational Mapping and Schema updates

Mautic is using ORM (Object Relational Mapping) as a way for the PHP code to relate to the MySQL database structure - as described here this is offered by the Symfony DoctrineBundle which integrates with a PHP tool called Doctrine.

Mapping information is essentially nothing but "metadata". It is a collection of rules that informs Doctrine ORM exactly how specific PHP classes and their properties are mapped to a specific database table in MySQL. The Mautic source code contains mapping information defining how the database structure is supposed to look for this version of Mautic (i.e. “Mautic 3.1 should have these tables with these features and these columns”). However, since Mautic is always being developed and improved, sometimes an upgrade to a newer version of Mautic needs to change things in the database. Since this is where all our data is stored, this should be handled with extreme care (we don’t want an update to accidentally corrupt the database!).

(Mautic uses* Semantic Versioning*, meaning that database changes that might break backwards compatibility are only introduced in* ***major*** *new versions - Mautic 3.0, Mautic 4.0, Mautic 5.0 etc., while database changes that add something new may also be introduced in* ***minor*** *new versions - Mautic 3.1, Mautic 3.2, etc.)*

Now, the common way to handle database changes is that, for example, the Mautic mapping information dictates 'The table called *X* should have a column called *Y* and then, when the PHP command *doctrine:schema:update* is run, Doctrine ORM notices that the MySQL table X is missing the required column Y and thus, it creates it in the database. If you run *doctrine:schema:update --dump-sql* you can see what SQL queries the Doctrine ORM believes need to be run in order to bring the actual MySQL database to match the mapping information provided by Mautic (the *--dump-sql* flag means “Do not actually do anything - just output what you would do”).

The DoctrineMigrationsBundle

So far so good. However, when the Mautic developers add or change functionality that they know requires changes to the database, they don’t just want to rely on the ORM being able to figure out what needs to change on the fly - your MySQL software or server might be slightly different than mine, and all of a sudden the ORM might behave differently on your Mautic than on mine, and support becomes much harder. Instead, developers can package the required SQL queries for a new change into a migration, letting everyone with access to the source code have an easy way of knowing what was altered, and making sure that the same change is applied consistently to everyone's database when upgrading.

This is where the Symfony DoctrineMigrationsBundle comes in. Instead of having the ORM just generating and firing off SQL queries blindly, the MigrationsBundle reads migrations from migration files - these are PHP files that usually contain functions like **up()** and **down()**, the former of which contains the SQL queries to be run when the migration is applied and the latter of which contains the 'reverse' SQL queries to be run if the migration is ever reversed - thus enabling an easier way to get out of situations where an SQL query crashed something unexpectedly. The migration files can also contain functions like **preUp()** and **postUp()** describing things to do immediately before or after applying the queries in the **up()** function.

(It is worth noting that most, if not all, Mautic-related migrations do not provide a down(). The reason for this is that since the migrations are applied alongside actual code upgrades of Mautic itself during upgrade, if you ever upgraded Mautic (+ applied the related migrations), but then later reverted some of the migrations, the Mautic code would get thoroughly confused.)*

Each migration is stored in its own file in the folder *app/migrations*. If you are running Mautic 2.x, first of all: please upgrade :), and secondly: you will most likely have 134 migration files in that folder, dating from the period 2015-2018. When Mautic 3 was introduced, the team got rid of a lot of the old migration files (since we know they would already have been applied during the upgrade process if people upgraded from 2.16.3 - and if people started with a new server from scratch with Mautic 3, hey, there would be nothing to migrate anyway). So in the Mautic 3.x codebase there are currently way fewer migration files available.

Whenever you run *doctrine:migrations:status*, the code reads the files in *app/migrations* and counts them as *Available Migrations*. Alongside this, it will read the table migrations in the MySQL database, which shows what migrations you have already applied (either through ordinary Mautic upgrade or, in rare cases of troubleshooting, manually with *doctrine:migrations:migrate*). Any migration files that do not show up in the *migrations* table are labeled as *New Migration* and will be applied if you ever run *doctrine:migrations:migrate*

(Note that the migration files themselves also contain checks to see if they have already been applied, so even if you run some old migrations the first time after, say, a fresh install, you might often see Schema includes this migration letting you know that the migration wasn't needed. It still counts as executed).*

That accounts for three of the numbers output by *doctrine:migrations:status - Executed Migrations, Available Migrations and New Migrations*. The command also outputs a fourth number: *Executed Unavailable Migrations* (with a scary red color). What's that? Simply put, it is the reverse logic of *New Migrations*:

- **New Migrations** include all migrations that have a migration file in **app/migrations**, but no corresponding row in the table **migrations** in the MySQL DB.
- **Executed Unavailable Migrations** include all migrations that have a row in **migration** in the MySQL DB but no corresponding migration file in **app/migrations**.

Despite the red color, it is rarely something to be concerned about. It is just the MigrationsBundle saying "Apparently we have applied a migration called Versionxxxxxx, but I have no idea what that migration was because I cannot find the corresponding migration file". It is fixed by making sure the number of rows in the *migrations* table and the number of files in _app/migration_s line up (and are named the same, obviously).

Forcibly restoring the Schema (as a last resort)

Keep in mind that except when DoctrineMigrationsBundle applies migrations, it actually has no idea what the database itself actually looks like. Thus, it is STRONGLY discouraged to manually alter stuff in the MySQL database on your own as you might end up in trouble down the road. As the Update Failed troubleshooting page describes, if your update ever fails and you are in trouble (and have taken a backup already) in descending order of importance you could try to:

1. Try the update script again - and let Mautic handle everything
2. Apply any outstanding migrations with **doctrine:migrations:migrate**
3. Let the ORM update the Schema with **doctrine:schema:update --force**

If you ever managed to make some bad, custom changes to the database and just want to have it return to the 'vanilla' Mautic structure, you can try running *doctrine:schema:update --force --complete*, which tells the Doctrine ORM to throw out anything in the database which doesn't match the mapping information provided by Mautic. This will cause any custom fields, tables, indexes etc. that you added directly in the database (as opposed to in the Mautic interface) to be dropped, as well as altering everything else to match the mapping information provided, while preserving your data. Keep in mind that this will (ironically) cause the migrations table to be dropped (as it is not part of the Mautic mapping information), and thus, the next time you run *doctrine:migrations:status*, all migration files in *app/migrations* will be considered *New Migrations* - even if applying them won't actually do anything.

Thus, running *doctrine:schema:update --force --complete* followed by *doctrine:migrations:migrate* should theoretically bring you back to a vanilla Mautic database schema with your data intact (except for data in whatever custom columns that potentially got dropped). Obviously, you should take plenty of backups before attempting this as this is very much a 'last resort'.

In conclusion

As mentioned, most users will never have to deal with any of this, as all the related transactions are handled by the automated Mautic upgrade script, but if you ever find yourself trying to troubleshoot a failed Mautic upgrade, hopefully you now have a pretty good idea of what the commands **doctrine:migrations:migrate** and **doctrine:schema:update** actually do.

Additional reading

- The Doctrine Project
- The Doctrine Project - Migrations
- Symfony - Databases and the Doctrine ORM
- Symfony - DoctrineBundle
- Symfony - DoctrineMigrationsBundle

Online URL:
https://kb.mautic.org/article/understanding-migrations-and-schema-update-with-doctrine-in-mautic.html