Doctrine Migrations

```
composer require doctrine/migrations
```

https://github.com/doctrine/migrations/

Introduction

The Doctrine Migrations project offers additional functionality on top of the DBAL and ORM for versioning your database schema. It makes it easy and safe to deploy changes to it in a way that can be reviewed and tested before being deployed to production.

Installation

You can use the Doctrine Migrations project by installing it with Composer or by downloading the latest PHAR from the releases page on GitHub.

For this documentation exercise we will assume you are starting a new project so create a new folder to work in:

```
mkdir /data/doctrine/migrations-docs-example
```

```
cd /data/doctrine/migrations-docs-example
```

Composer

Now to install with Composer it is as simple as running the following command in your project.

```
composer require "doctrine/migrations"
```

Now you will have a file in `vendor/bin` available to run the migrations console application:

```
./vendor/bin/doctrine-migrations
```

PHAR

To install by downloading the PHAR, you just need to download the latest PHAR file from the releases page on GitHub.

Here is an example using the `2.0.0` release:

```
wget https://github.com/doctrine/migrations/releases/download/v2.0.0/doctrine-
migrations.phar
```

Now you can execute the PHAR like this:

```
php doctrine-migrations.phar
```

Configuration

So you are ready to start configuring your migrations? We just need to provide a few bits of information for the console application in order to get started.

Migrations Configuration

First we need to configure information about your migrations. In `/data/doctrine/migrations-docs-example` go ahead and create a folder to store your migrations in:

```
mkdir -p lib/MyProject/Migrations
```

Now, in the root of your project place a file named `migrations.php`, `migrations.yml`, `migrations.xml` or `migrations.json` and place the following contents:

- *PHP*

```
<?php

return [
    'table_storage' => [
        'table_name' => 'doctrine_migration_versions',
        'version_column_name' => 'version',
        'version_column_length' => 1024,
        'executed_at_column_name' => 'executed_at',
        'execution_time_column_name' => 'execution_time',
    ],

    'migrations_paths' => [
        'MyProject\Migrations' => '/data/doctrine/migrations/lib/MyProject/Migrations',
        'MyProject\Component\Migrations' => './Component/MyProject/Migrations',
    ],

    'all_or_nothing' => true,
    'check_database_platform' => true,
    'organize_migrations' => 'none',
];
```

- *YAML*
- *XML*
- *JSON*

Please note that if you want to use the YAML configuration option, you will need to install the `symfony/yaml` package with composer:

```
composer require symfony/yaml
```

Here are details about what each configuration option does:

Name	Required	Default	Description
migrations_paths< string, st	ing> |	yes | null	The PHP namespace your migration classes are located under and the path to a directory where to look for

Name	Required	Default	Description
			migration classes.
table_storage	no		Used by doctrine migrations to track the currently executed migrations
all_or_nothing	no	false	Whether or not to wrap multiple migrations in a single transaction.
migrations	no	[]	Manually specify the array of migration versions instead of finding migrations.
check_database_platform	no	true	Whether to add a database platform check at the beginning of the generated code.
organize_migrations	no	`none`	Whether to organize migration classes under year (`year`) or year and month (`year_and_month`) subdirectories.

Here the possible options for `table_storage`:

Name	Required	Default	Description
table_name	no	doctrine_migration_versions	The name of the table to track executed migrations in.
version_column_name	no	version	The name of the column which stores the version name.
version_column_length	no	1024	The length of the column which stores the version name.
executed_at_column_name	no	executed_at	The name of the column which stores the date that a migration was executed.
execution_time_column_name	no	execution_time	The name of the column which stores how long a migration took (milliseconds).

Manually Providing Migrations

If you don't want to rely on Doctrine finding your migrations, you can explicitly specify the array of migration classes using the `migrations` configuration setting:

- *PHP*

```
<?php

return [
    // ..

    'migrations' => [
        'MyProject\Migrations\NewMigration',
    ],
];
```

- *YAML*
- *XML*
- *JSON*

All or Nothing Transaction

This only works if your database supports transactions for DDL statements.

When using the `all_or_nothing` option, multiple migrations ran at the same time will be wrapped in a single transaction. If one migration fails, all migrations will be rolled back

From the Command Line

You can also set this option from the command line with the `migrate` command and the `--all-or-nothing` option:

```
./vendor/bin/doctrine-migrations migrate --all-or-nothing
```

If you have it enabled at the configuration level and want to change it for an individual migration you can pass a value of `0` or `1` to `--all-or-nothing`.

```
./vendor/bin/doctrine-migrations migrate --all-or-nothing=0
```

Connection Configuration

Now that we've configured our migrations, the next thing we need to configure is how the migrations console application knows how to get the connection to use for the migrations:

Simple

The simplest configuration is to put a `migrations-db.php` file in the root of your project and return an array of connection information that can be passed to the DBAL:

```
<?php

return [
    'dbname' => 'migrations_docs_example',
    'user' => 'root',
    'password' => '',
    'host' => 'localhost',
    'driver' => 'pdo_mysql',
];
```

You will need to make sure the `migrations_docs_example` database exists. If you are using MySQL you can create it with the following command:

```
mysqladmin create migrations_docs_example
```

If you have already a DBAL connection available in your application, `migrations-db.php` can return it directly:

```
<?php
use Doctrine\DBAL\DriverManager;

return DriverManager::getConnection([
    'dbname' => 'migrations_docs_example',
    'user' => 'root',
    'password' => '',
    'host' => 'localhost',
    'driver' => 'pdo_mysql',
]);
```

Advanced

If you require a more advanced configuration and you want to get the connection to use from your existing application setup then you can use this method of configuration.

In the root of your project, place a file named `cli-config.php` with the following contents. It can also be placed in a folder named `config` if you prefer to keep it out of the root of your project.

```
<?php

require 'vendor/autoload.php';

use Doctrine\DBAL\DriverManager;
use Doctrine\Migrations\Configuration\Configuration\PhpFile;
use Doctrine\Migrations\Configuration\Connection\ExistingConnection;
use Doctrine\Migrations\DependencyFactory;

$config = new PhpFile('migrations.php'); // Or use one of the Doctrine\Migrations\Configuration\
Configuration\* loaders

$conn = DriverManager::getConnection(['driver' => 'pdo_sqlite', 'memory' => true]);

return DependencyFactory::fromConnection($config, new ExistingConnection($conn));
```

The above setup assumes you are not using the ORM. If you want to use the ORM, first require it in your project with composer:

```
composer require doctrine/orm
```

Now update your `cli-config.php` in the root of your project to look like the following:

```
<?php

require 'vendor/autoload.php';

use Doctrine\ORM\EntityManager;
use Doctrine\ORM\Tools\Setup;
use Doctrine\Migrations\Configuration\EntityManager\ExistingEntityManager;
use Doctrine\Migrations\DependencyFactory;

$config = new PhpFile('migrations.php'); // Or use one of the Doctrine\Migrations\Configuration\
Configuration\* loaders

$paths = [__DIR__.'/lib/MyProject/Entities'];
$isDevMode = true;

$ORMconfig = Setup::createAnnotationMetadataConfiguration($paths, $isDevMode);
$entityManager = EntityManager::create(['driver' => 'pdo_sqlite', 'memory' => true],
$ORMconfig);

return DependencyFactory::fromEntityManager($config, new
ExistingEntityManager($entityManager));
```

Make sure to create the directory where your ORM entities will be located:

```
mkdir lib/MyProject/Entities
```

Migration Classes

Migration classes must extend `Doctrine\Migrations\AbstractMigration` and at a minimum they must implement the `up` and `down` methods. You can easily generate a blank migration to modify with the following command:

```
./vendor/bin/doctrine-migrations generate
Generated new migration class to
"/data/doctrine/migrations-docs-example/lib/MyProject/Migrations/Version20180601
193057.php"

To run just this migration for testing purposes, you can use migrations:execute
--up 'MyProject\Migrations\Version20180601193057'

To revert the migration you can use migrations:execute --down 'MyProject\
Migrations\Version20180601193057'
```

The above command will generate a PHP class with the path to it visible like above. Here is what the blank migration looks like:

```
<?php

declare(strict_types=1);

namespace MyProject\Migrations;

use Doctrine\DBAL\Schema\Schema;
use Doctrine\Migrations\AbstractMigration;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20180601193057 extends AbstractMigration
{
    public function getDescription() : string
    {
        return '';
    }

    public function up(Schema $schema) : void
    {
        // this up() migration is auto-generated, please modify it to your needs

    }

    public function down(Schema $schema) : void
    {
        // this down() migration is auto-generated, please modify it to your needs

    }
}
```

Methods to Implement

The `AbstractMigration` class provides a few methods you can override to define additional behavior for the migration.

isTransactional

Override this method if you want to disable transactions in a migration. It defaults to true.

```
public function isTransactional() : bool
{
    return false;
}
```

Some database platforms like MySQL or Oracle do not support DDL statements in transactions and may or may not implicitly commit the transaction opened by this library as soon as they encounter such a statement, and before running it. Make sure to read the manual of your database platform to know what is actually happening. `isTransactional()` does not guarantee that statements are wrapped in a single transaction.

getDescription

Override this method if you want to provide a description for your migration. The value returned here will get outputted when you run the `./vendor/bin/doctrine-migrations status --show-versions` command.

```
public function getDescription() : string
{
   return 'The description of my awesome migration!';
}
```

preUp

This method gets called before the `up()` is called.

```
public function preUp(Schema $schema) : void
{
}
```

postUp

This method gets called after the `up()` is called.

```
public function postUp(Schema $schema) : void
{
}
```

preDown

This method gets called before the `down()` is called.

```
public function preDown(Schema $schema) : void
```

```
{
}
```

postDown

This method gets called after the `down()` is called.

```
public function postDown(Schema $schema) : void
{
}
```

Methods to Call

The `AbstractMigration` class provides a few methods you can call in your migrations to perform various functions.

warnIf

Warn with a message if some condition is met.

```
public function up(Schema $schema) : void
{
    $this->warnIf(true, 'Something might be going wrong');

    // ...
}
```

abortIf

Abort the migration if some condition is met:

```
public function up(Schema $schema) : void
{
    $this->abortIf(true, 'Something went wrong. Aborting.');

    // ...
}
```

skipIf

Skip the migration if some condition is met.

```
public function up(Schema $schema) : void
{
    $this->skipIf(true, 'Skipping this migration.');

    // ...
}
```

addSql

You can use the `addSql` method within the `up` and `down` methods. Internally the `addSql` calls are passed to the executeQuery method in the DBAL. This means that you can use the power of prepared statements easily and that you don't need to copy paste the same query with different parameters. You can just pass those different parameters to the addSql method as parameters.

```
public function up(Schema $schema) : void
{
    $users = [
        ['name' => 'mike', 'id' => 1],
        ['name' => 'jwage', 'id' => 2],
        ['name' => 'ocramius', 'id' => 3],
    ];

    foreach ($users as $user) {
        $this->addSql('UPDATE user SET happy = true WHERE name = :name AND id = :id', $user);
    }
}
```

write

Write some debug information to the console.

```
public function up(Schema $schema) : void
{
    $this->write('Doing some cool migration!');

    // ...
}
```

throwIrreversibleMigrationException

If a migration cannot be reversed, you can use this exception in the `down` method to indicate such. The `throwIrreversibleMigrationException` method accepts an optional message to output.

```
public function down(Schema $schema) : void
{
    $this->throwIrreversibleMigrationException();

    // ...
}
```

Managing Migrations

Managing migrations with Doctrine is easy. You can execute migrations from the console and easily revert them. You also have the option to write the SQL for a migration to a file instead of executing it from PHP.

Status

Now that we have a new migration created, run the `status` command with the `--show-versions` option to see that the new migration is registered and ready to be executed:

```
./vendor/bin/doctrine-migrations status --show-versions

 == Configuration

    >> Name:                                              My Project Migrations
    >> Database Driver:                                   pdo_mysql
    >> Database Host:                                     localhost
    >> Database Name:
migrations_docs_example
    >> Configuration Source:
/data/doctrine/migrations-docs-example/migrations.php
    >> Version Table Name:
doctrine_migration_versions
    >> Version Column Name:                               version
    >> Migrations Namespace:                              MyProject\Migrations
    >> Migrations Directory:
/data/doctrine/migrations-docs-example/lib/MyProject/Migrations
    >> Previous Version:                                  Already at first
version
    >> Current Version:                                   0
    >> Next Version:                                      2018-06-01 19:30:57
(MyProject\Migrations\Version20180601193057)
    >> Latest Version:                                    2018-06-01 19:30:57
(MyProject\Migrations\Version20180601193057)
    >> Executed Migrations:                               0
    >> Executed Unavailable Migrations:                   0
    >> Available Migrations:                              1
    >> New Migrations:                                    1

 == Available Migration Versions

    >> 2018-06-01 19:30:57 (MyProject\Migrations\Version20180601193057) not
migrated     This is my example migration.
```

As you can see we have a new migration version available and it is ready to be executed. The problem is, it does not have anything in it so nothing would be executed! Let's add some code to it and add a new table:

```
<?php

declare(strict_types=1);

namespace MyProject\Migrations;

use Doctrine\DBAL\Schema\Schema;
use Doctrine\Migrations\AbstractMigration;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20180601193057 extends AbstractMigration
{
    public function getDescription() : string
    {
        return 'This is my example migration.';
    }

    public function up(Schema $schema) : void
    {
        $this->addSql('CREATE TABLE example_table (id INT AUTO_INCREMENT NOT NULL,
title VARCHAR(255) DEFAULT NULL, PRIMARY KEY(id))');
    }

    public function down(Schema $schema) : void
    {
        $this->addSql('DROP TABLE example_table');
    }
}
```

Dry Run

Now we are ready to give it a test! First lets just do a dry-run to make sure it produces the SQL we expect:

```
./vendor/bin/doctrine-migrations migrate --dry-run

                    My Project Migrations


Executing dry run of migration up to MyProject\Migrations\Version20180601193057
from 0

  ++ migrating MyProject\Migrations\Version20180601193057

     -> CREATE TABLE example_table (id INT AUTO_INCREMENT NOT NULL, title
VARCHAR(255) DEFAULT NULL, PRIMARY KEY(id))

  ++ migrated (took 60.9ms, used 8M memory)

  ------------------------

  ++ finished in 69.4ms
  ++ used 8M memory
  ++ 1 migrations executed
  ++ 1 sql queries
```

Executing Multiple Migrations

Everything looks good so we can remove the `--dry-run` option and actually execute the migration.

> The `migrate` command will execute multiple migrations if there are multiple new unexecuted migration versions available. It will attempt to go from the current version to the latest version available.

```
./vendor/bin/doctrine-migrations migrate
```

```
                    My Project Migrations


WARNING! You are about to execute a database migration that could result in
schema changes and data loss. Are you sure you wish to continue? (y/n)y
Migrating up to MyProject\Migrations\Version20180601193057 from 0

  ++ migrating MyProject\Migrations\Version20180601193057

     -> CREATE TABLE example_table (id INT AUTO_INCREMENT NOT NULL, title
VARCHAR(255) DEFAULT NULL, PRIMARY KEY(id))

  ++ migrated (took 47.7ms, used 8M memory)

  ------------------------

  ++ finished in 49.1ms
  ++ used 8M memory
  ++ 1 migrations executed
  ++ 1 sql queries
```

Executing Single Migrations

You may want to just execute a single migration up or down. You can do this with the `execute` command:

```
./vendor/bin/doctrine-migrations execute MyProject\Migrations\
Version20180601193057 --down
WARNING! You are about to execute a database migration that could result in
schema changes and data lost. Are you sure you wish to continue? (y/n)y

  ++ migrating MyProject\Migrations\Version20180601193057

     -> DROP TABLE example_table

  ++ migrated (took 42.6ms, used 8M memory)
```

No Interaction

Alternately, if you wish to run the migrations in an unattended mode, we can add the `--no-interaction` option and then execute the migrations without any extra prompting from Doctrine.

```
./vendor/bin/doctrine-migrations migrate --no-interaction

                    My Project Migrations


Migrating up to MyProject\Migrations\Version20180601193057 from 0

  ++ migrating MyProject\Migrations\Version20180601193057

     -> CREATE TABLE example_table (id INT AUTO_INCREMENT NOT NULL, title
VARCHAR(255) DEFAULT NULL, PRIMARY KEY(id))

  ++ migrated (took 46.5ms, used 8M memory)

  ------------------------
  ++ finished in 47.3ms
  ++ used 8M memory
  ++ 1 migrations executed
  ++ 1 sql queries
```

By checking the status again after using either method you will see everything is updated:

```
./vendor/bin/doctrine-migrations status --show-versions

 == Configuration

    >> Name:                                               My Project Migrations
    >> Database Driver:                                    pdo_mysql
    >> Database Host:                                      localhost
    >> Database Name:
migrations_docs_example
    >> Configuration Source:
/data/doctrine/migrations-docs-example/migrations.php
    >> Version Table Name:
doctrine_migration_versions
    >> Version Column Name:                                version
    >> Migrations Namespace:                               MyProject\Migrations
    >> Migrations Directory:
/data/doctrine/migrations-docs-example/lib/MyProject/Migrations
    >> Previous Version:                                   0
    >> Current Version:                                    2018-06-01 19:30:57
(MyProject\Migrations\Version20180601193057)
    >> Next Version:                                       Already at latest
version
    >> Latest Version:                                     2018-06-01 19:30:57
(MyProject\Migrations\Version20180601193057)
    >> Executed Migrations:                                1
    >> Executed Unavailable Migrations:                    0
    >> Available Migrations:                               1
    >> New Migrations:                                     0

 == Available Migration Versions

    >> 2018-06-01 19:30:57 (MyProject\Migrations\Version20180601193057)
migrated (executed at 2018-06-01 17:08:44)     This is my example migration.
```

Reverting Migrations

The `migrate` command optionally accepts a version or version alias to migrate to. By default it will try to migrate up from the current version to the latest version. If you pass a version that is older than the current version, it will migrate down. To rollback to the the first version you can use the `first` version alias:

```
./vendor/bin/doctrine-migrations migrate first

                    My Project Migrations


WARNING! You are about to execute a database migration that could result in
schema changes and data loss. Are you sure you wish to continue? (y/n)y
Migrating down to 0 from MyProject\Migrations\Version20180601193057

  -- reverting MyProject\Migrations\Version20180601193057

     -> DROP TABLE example_table
  -- reverted (took 38.4ms, used 8M memory)

  ------------------------

  ++ finished in 39.5ms
  ++ used 8M memory
  ++ 1 migrations executed
  ++ 1 sql queries
```

Now if you run the `status` command again, you will see that the database is back to the way it was before:

```
./vendor/bin/doctrine-migrations status --show-versions

 == Configuration

    >> Name:                                               My Project Migrations
    >> Database Driver:                                    pdo_mysql
    >> Database Host:                                      localhost
    >> Database Name:
migrations_docs_example
    >> Configuration Source:
/data/doctrine/migrations-docs-example/migrations.php
    >> Version Table Name:
doctrine_migration_versions
    >> Version Column Name:                                version
    >> Migrations Namespace:                               MyProject\Migrations
    >> Migrations Directory:
/data/doctrine/migrations-docs-example/lib/MyProject/Migrations
    >> Previous Version:                                   Already at first
version
    >> Current Version:                                    0
    >> Next Version:                                       2018-06-01 19:30:57
(MyProject\Migrations\Version20180601193057)
    >> Latest Version:                                     2018-06-01 19:30:57
(MyProject\Migrations\Version20180601193057)
    >> Executed Migrations:                                0
    >> Executed Unavailable Migrations:                    0
    >> Available Migrations:                               1
    >> New Migrations:                                     1

 == Available Migration Versions

    >> 2018-06-01 19:30:57 (MyProject\Migrations\Version20180601193057)
not migrated     This is my example migration.
```

Version Aliases

You can use version aliases when executing migrations. This is for your convenience so you don't have to always know the version number. The following aliases are available:

- `first` - Migrate down to before the first version.
- `prev` - Migrate down to before the previous version.
- `next` - Migrate up to the next version.
- `latest` - Migrate up to the latest version.

Here is an example where we migrate to the latest version and then revert back to the first:

```
./vendor/bin/doctrine-migrations migrate latest
```

```
./vendor/bin/doctrine-migrations migrate first
```

Writing Migration SQL Files

You can optionally choose to not execute a migration directly on a database from PHP and instead output all the SQL statement to a file. This is possible by using the `--write-sql` option:

```
./vendor/bin/doctrine-migrations migrate --write-sql

                    My Project Migrations


Executing dry run of migration up to MyProject\Migrations\Version20180601193057
from 0

  ++ migrating MyProject\Migrations\Version20180601193057

     -> CREATE TABLE example_table (id INT AUTO_INCREMENT NOT NULL, title
VARCHAR(255) DEFAULT NULL, PRIMARY KEY(id))

  ++ migrated (took 55ms, used 8M memory)

  ------------------------

  ++ finished in 60.7ms
  ++ used 8M memory
  ++ 1 migrations executed
  ++ 1 sql queries
-- Migrating from 0 to MyProject\Migrations\Version20180601193057


Writing migration file to
"/data/doctrine/migrations-docs-example/doctrine_migration_20180601172528.sql"
```

Now if you have a look at the `doctrine_migration_20180601172528.sql` file you will see the would be executed SQL outputted in a nice format:

```
cat doctrine_migration_20180601172528.sql
-- Doctrine Migration File Generated on 2018-06-01 17:25:28

-- Version MyProject\Migrations\Version20180601193057
CREATE TABLE example_table (id INT AUTO_INCREMENT NOT NULL, title VARCHAR(255)
DEFAULT NULL, PRIMARY KEY(id));
INSERT INTO doctrine_migration_versions (version, executed_at) VALUES
('MyProject\Migrations\Version20180601193057', CURRENT_TIMESTAMP);
```

The `--write-sql` option also accepts an optional value for where to write the sql file. It can be a relative path to a file that will write to the current working directory:

```
./vendor/bin/doctrine-migrations migrate --write-sql=migration.sql
```

Or it can be an absolute path to the file:

```
./vendor/bin/doctrine-migrations migrate --write-sql=/path/to/migration.sql
```

Or it can be a directory and it will write the default filename to it:

```
./vendor/bin/doctrine-migrations migrate --write-sql=/path/to/directory
```

Managing the Version Table

Sometimes you may need to manually mark a migration as migrated or not. You can use the `version` command for this.

> Use caution when using the `version` command. If you delete a version from the table and then run the `migrate` command, that migration version will be executed again.

```
./vendor/bin/doctrine-migrations version 'MyProject\Migrations\
Version20180601193057' --add
```

Or you can delete that version:

```
./vendor/bin/doctrine-migrations version 'MyProject\Migrations\
Version20180601193057' --delete
```

This command does not actually execute any migrations, it just adds or deletes the version from the version table where we track whether or not a migration version has been executed or not.

Generating Migrations

Doctrine can generate blank migrations for you to modify or it can generate functional migrations for you by comparing the current state of your database schema to your mapping information.

Generating Blank Migrations

To generate a blank migration you can use the `generate` command:

```
./vendor/bin/doctrine-migrations generate
```

Diffing Using the ORM

If you are using the ORM, you can modify your mapping information and have Doctrine generate a migration for you by comparing the current state of your database schema to the mapping information that is defined by using the ORM. To test this functionality, create a new `User` entity located at `lib/MyProject/Entities/User.php`.

```
<?php

namespace MyProject\Entities;

/**
 * @Entity
 * @Table(name="users")
 */
class User
{
    /** @Id @Column(type="integer") @GeneratedValue */
    private $id;

    /** @Column(type="string", nullable=true) */
    private $username;

    public function setId(int $id)
    {
        $this->id = $id;
    }

    public function getId() : ?int
    {
        return $this->id;
    }

    public function setUsername(string $username) : void
    {
        $this->username = $username;
    }

    public function getUsername() : ?string
    {
        return $this->username;
    }
}
```

Now when you run the `diff` command it will generate a migration which will create the `users` table:

```
./vendor/bin/doctrine-migrations diff
Generated new migration class to
"/data/doctrine/migrations-docs-example/lib/MyProject/Migrations/Version20180601
215504.php"

To run just this migration for testing purposes, you can use migrations:execute
--up 'MyProject\Migrations\Version20180601215504'

To revert the migration you can use migrations:execute --down 'MyProject\
Migrations\Version20180601215504'
```

Take a look at the generated migration:

> Notice how the table named `example_table` that we created earlier in the Managing Migrations chapter is being dropped. This is because the table is not mapped anywhere in the Doctrine ORM and the `diff` command detects that and generates the SQL to drop the table. If you want to ignore some tables in your database take a look at Ignoring Custom Tables chapter.

```
<?php

declare(strict_types=1);

namespace MyProject\Migrations;
```

```
use Doctrine\DBAL\Schema\Schema;
use Doctrine\Migrations\AbstractMigration;

/**
 * Auto-generated Migration: Please modify to your needs!
 */
final class Version20180601215504 extends AbstractMigration
{
    public function getDescription() : string
    {
        return '';
    }

    public function up(Schema $schema) : void
    {
        // this up() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration
can only be executed safely on \'mysql\'.');

        $this->addSql('CREATE TABLE users (id INT AUTO_INCREMENT NOT NULL, username
VARCHAR(255) DEFAULT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8
COLLATE utf8_unicode_ci ENGINE = InnoDB');
        $this->addSql('DROP TABLE example_table');
    }

    public function down(Schema $schema) : void
    {
        // this down() migration is auto-generated, please modify it to your needs
        $this->abortIf($this->connection->getDatabasePlatform()->getName() !== 'mysql', 'Migration
can only be executed safely on \'mysql\'.');

        $this->addSql('CREATE TABLE example_table (id INT AUTO_INCREMENT NOT NULL,
title VARCHAR(255) DEFAULT NULL COLLATE latin1_swedish_ci, PRIMARY KEY(id))
DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci ENGINE = InnoDB');
        $this->addSql('DROP TABLE users');
    }
}
```

Now you are ready to execute your diff migration:

```
./vendor/bin/doctrine-migrations migrate

                    My Project Migrations


WARNING! You are about to execute a database migration that could result in
schema changes and data loss. Are you sure you wish to continue? (y/n)y
Migrating up to MyProject\Migrations\Version20180601215504 from MyProject\
Migrations\Version20180601193057

  ++ migrating MyProject\Migrations\Version20180601215504
```

```
     -> CREATE TABLE users (id INT AUTO_INCREMENT NOT NULL, username
VARCHAR(255) DEFAULT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8 COLLATE
utf8_unicode_ci ENGINE = InnoDB
     -> DROP TABLE example_table

  ++ migrated (took 75.9ms, used 8M memory)

  ------------------------

  ++ finished in 84.3ms
  ++ used 8M memory
  ++ 1 migrations executed
  ++ 1 sql queries
```

The SQL generated here is the exact same SQL that would be executed if you were using the `orm:schema-tool` command. This just allows you to capture that SQL and maybe tweak it or add to it and trigger the deployment later across multiple database servers.

Diffing Without the ORM

Internally the diff command generates a `Doctrine\DBAL\Schema\Schema` object from your entities metadata using an implementation of `Doctrine\Migrations\Provider\SchemaProviderInterface`. To use the Schema representation directly, without the ORM, you must implement this interface yourself.

The `SchemaProviderInterface` only has one method named `createSchema`. This should return a `Doctrine\DBAL\Schema\Schema` instance that represents the state to which you'd like to migrate your database.

```
<?php

use Doctrine\DBAL\Schema\Schema;
use Doctrine\Migrations\Provider\SchemaProviderInterface;

final class CustomSchemaProvider implements SchemaProviderInterface
{
    public function createSchema()
    {
        $schema = new Schema();

        $table = $schema->createTable('users');

        $table->addColumn('id', 'integer', [
            'autoincrement' => true,
        ]);

        $table->addColumn('username', 'string', [
            'notnull' => false,
        ]);

        $table->setPrimaryKey(array('id'));
        return $schema;
    }
}
```

The `StubSchemaProvider` provided with the migrations library is another option. It simply takes a schema object to its constructor and returns it from `createSchema`.

```
<?php

use Doctrine\DBAL\Schema\Schema;
use Doctrine\Migrations\Provider\StubSchemaProvider;

$schema = new Schema();

$table = $schema->createTable('users');

$table->addColumn('id', 'integer', [
    'autoincrement' => true,
]);

$table->addColumn('username', 'string', [
    'notnull' => false,
]);

$table->setPrimaryKey(array('id'));

$provider = new StubSchemaProvider($schema);
$provider->createSchema() === $schema; // true
```

By default the Doctrine Migrations command line tool will only add the diff command if the ORM is present. Without the ORM, you'll have to add the diff command to your console application manually, passing in your schema provider implementation to the diff command's constructor. Take a look at the Custom Integration chapter for information on how to setup a custom console application.

```
<?php

use Doctrine\Migrations\Tools\Console\Command\DiffCommand;

$schemaProvider = new CustomSchemaProvider();

/** @var Symfony\Component\Console\Application */
$cli->add(new DiffCommand($schemaProvider));

// ...

$cli→run();
```

With the custom provider in place the `diff` command will compare the current database schema to the one provided by the `SchemaProviderInterface` implementation. If there is a mismatch, the differences will be included in the generated migration just like the ORM examples above.

Formatted SQL

You can optionally pass the `--formatted` option if you want the dumped SQL to be formatted. This option uses the `doctrine/sql-formatter` package so you will need to install this package for it to work:

```
composer require doctrine/sql-formatter
```

Ignoring Custom Tables

If you have custom tables which are not managed by Doctrine you will need to tell Doctrine to ignore these tables. Otherwise, everytime you run the `diff` command, Doctrine will try to drop those tables. You can configure Doctrine with a schema filter.

```
$connection->getConfiguration()->setFilterSchemaAssetsExpression("~^(?!t_)~");
```

With this expression all tables prefixed with t will ignored by the schema tool.

If you use the DoctrineBundle with Symfony you can set the `schema_filter` option in your configuration. You can find more information in the documentation of the DoctrineMigrationsBundle.

Merging Historical Migrations

If you have many migrations, which were generated by successive runs of the `diff` command over time, and you would like to replace them with one single migration, you can delete (or archive) all your historical migration files and run the `diff` command with the `--from-empty-schema` option. It will generate a full migration as if your database was empty. You can then use the `rollup` command to synchronize the version table of your (already up-to-date) database.

Custom Configuration

It is possible to build a custom configuration where you manually build the `Doctrine\Migrations\Configuration\Configuration` instance instead of using YAML, XML, etc. In order to do this, you will need to setup a Custom Integration.

Once you have your custom integration setup, you can modify it to look like the following:

```
#!/usr/bin/env php
<?php

require_once __DIR__.'/vendor/autoload.php';

use Doctrine\DBAL\DriverManager;
use Doctrine\Migrations\Configuration\Configuration;
use Doctrine\Migrations\Configuration\Connection\ExistingConnection;
use Doctrine\Migrations\Configuration\Configuration\ExistingConfiguration;
use Doctrine\Migrations\DependencyFactory;
use Doctrine\Migrations\Metadata\Storage\TableMetadataStorageConfiguration;
use Doctrine\Migrations\Tools\Console\Command;
use Symfony\Component\Console\Application;

$dbParams = [
   'dbname' => 'migrations_docs_example',
   'user' => 'root',
   'password' => '',
   'host' => 'localhost',
   'driver' => 'pdo_mysql',
];

$connection = DriverManager::getConnection($dbParams);

$configuration = new Configuration($connection);

$configuration->setName('My Project Migrations');
$configuration->addMigrationsDirectory('MyProject\Migrations', '/data/doctrine/migrations-docs-
example/lib/MyProject/Migrations');
$configuration->setAllOrNothing(true);
$configuration->setCheckDatabasePlatform(false);

$storageConfiguration = new TableMetadataStorageConfiguration();
$storageConfiguration->setTableName('doctrine_migration_versions');

$configuration->setMetadataStorageConfiguration($storageConfiguration);

$dependencyFactory = DependencyFactory::fromConnection(
   new ExistingConfiguration($configuration),
   new ExistingConnection($connection)
);

$cli = new Application('Doctrine Migrations');
$cli->setCatchExceptions(true);

$cli->addCommands(array(
   new Command\DumpSchemaCommand($dependencyFactory),
   new Command\ExecuteCommand($dependencyFactory),
   new Command\GenerateCommand($dependencyFactory),
   new Command\LatestCommand($dependencyFactory),
   new Command\ListCommand($dependencyFactory),
   new Command\MigrateCommand($dependencyFactory),
   new Command\RollupCommand($dependencyFactory),
   new Command\StatusCommand($dependencyFactory),
   new Command\SyncMetadataCommand($dependencyFactory),
   new Command\VersionCommand($dependencyFactory),
));

$cli→run();
```

Migrations Events

The Doctrine Migrations library emits a series of events during the migration process.

- `onMigrationsMigrating`: dispatched immediately before starting to execute versions. This does not fire if there are no versions to be executed.
- `onMigrationsVersionExecuting`: dispatched before a single version executes.
- `onMigrationsVersionExecuted`: dispatched after a single version executes.
- `onMigrationsVersionSkipped`: dispatched when a single version is skipped.
- `onMigrationsMigrated`: dispatched when all versions have been executed.

All of these events are emitted via the DBAL connection's event manager. Here's an example event subscriber that listens for all possible migrations events.

```
<?php

use Doctrine\Common\EventSubscriber;
use Doctrine\Migrations\Event\MigrationsEventArgs;
use Doctrine\Migrations\Event\MigrationsVersionEventArgs;
use Doctrine\Migrations\Events;

class MigrationsListener implements EventSubscriber
{
    public function getSubscribedEvents() : array
    {
        return [
            Events::onMigrationsMigrating,
            Events::onMigrationsMigrated,
            Events::onMigrationsVersionExecuting,
            Events::onMigrationsVersionExecuted,
            Events::onMigrationsVersionSkipped,
        ];
    }

    public function onMigrationsMigrating(MigrationsEventArgs $args) : void
    {
        // ...
    }

    public function onMigrationsMigrated(MigrationsEventArgs $args) : void
    {
        // ...
    }

    public function onMigrationsVersionExecuting(MigrationsVersionEventArgs $args) : void
    {
        // ...
    }

    public function onMigrationsVersionExecuted(MigrationsVersionEventArgs $args) : void
    {
        // ...
    }

    public function onMigrationsVersionSkipped(MigrationsVersionEventArgs $args) : void
    {
        // ...
    }
}
```

To add an event subscriber to a connections event manager, use the `Connection::getEventManager()` method and the `EventManager::addEventSubscriber()` method:

This might go in the `cli-config.php` file or somewhere in a frameworks container or dependency injection configuration.

```
<?php

use Doctrine\DBAL\DriverManager;

$connection = DriverManager::getConnection([
    // ...
]);

$connection->getEventManager()->addEventSubscriber(new MigrationsListener());

// rest of the cli set up…
```

Version Numbers

When Generating Migrations the newly created classes are generated with the name `Version{date}` with `{date}` having a `YmdHis`format. This format is important as it allows the migrations to be correctly ordered.

Starting with version 1.5 when loading migration classes, Doctrine does a `sort($versions, SORT_STRING)` on version numbers. This can cause problems with custom version numbers:

```
<?php

$versions = [
    'Version1',
    'Version2',
    // ...
    'Version10',
];

sort($versions, SORT_STRING);

var_dump($versions);

/*
```

```
array(3) {
  [0] =>
  string(8) "Version1"
  [1] =>
  string(9) "Version10"
  [2] =>
  string(8) "Version2"
}
*/
```

The custom version numbers above end up out of order which may cause damage to a database.

It is **strongly recommended** that the `Version{date}` migration class name format is used and that the various tools for generating migrations are used.

Should some custom migration numbers be necessary, keeping the version number the same length as the date format (14 total characters) and padding it to the left with zeros should work.

```
<?php

$versions = [
    'Version00000000000001',
    'Version00000000000002',
    // ...
    'Version00000000000010',
    'Version20180107070000', // generated version
];

sort($versions, SORT_STRING);

var_dump($versions);
/*
array(4) {
  [0] =>
  string(21) "Version00000000000001"
  [1] =>
  string(21) "Version00000000000002"
  [2] =>
  string(21) "Version00000000000010"
  [3] =>
  string(21) "Version20180107070000"
}
*/
```

Please note that migrating to this new, zero-padded format may require manual version table intervention if the versions have previously been applied.

Integrations

If you are using a framework, you can use one of the pre-existing integrations built by the community.

- Symfony
- Zend
- Laravel
- Silex
- Silex
- Nette

Don't hesitate to make a Pull Request if you want to add your integration to this list.

Custom Integration

If you don't want to use the `./vendor/bin/doctrine-migrations` script that comes with the project, you can always setup your own custom integration.

In the root of your project, create a file named `migrations` and make it executable:

```
chmod +x migrations
```

Now place the following code in the `migrations` file:

```
#!/usr/bin/env php
<?php

require_once __DIR__.'/vendor/autoload.php';

use Doctrine\DBAL\DriverManager;
use Doctrine\Migrations\DependencyFactory;
use Doctrine\Migrations\Configuration\Migration\PhpFile;
use Doctrine\Migrations\Configuration\Connection\ExistingConnection;
use Doctrine\Migrations\Tools\Console\Command;
use Symfony\Component\Console\Application;

$dbParams = [
    'dbname' => 'migrations_docs_example',
    'user' => 'root',
    'password' => '',
    'host' => 'localhost',
    'driver' => 'pdo_mysql',
];

$connection = DriverManager::getConnection($dbParams);

$config = new PhpFile('migrations.php'); // Or use one of the Doctrine\Migrations\Configuration\Configuration\* loaders

$dependencyFactory = DependencyFactory::fromConnection($config, new ExistingConnection($connection));

$cli = new Application('Doctrine Migrations');
$cli->setCatchExceptions(true);
```

```
$cli->addCommands(array(
    new Command\DumpSchemaCommand($dependencyFactory),
    new Command\ExecuteCommand($dependencyFactory),
    new Command\GenerateCommand($dependencyFactory),
    new Command\LatestCommand($dependencyFactory),
    new Command\ListCommand($dependencyFactory),
    new Command\MigrateCommand($dependencyFactory),
    new Command\RollupCommand($dependencyFactory),
    new Command\StatusCommand($dependencyFactory),
    new Command\SyncMetadataCommand($dependencyFactory),
    new Command\VersionCommand($dependencyFactory),
));

$cli→run();
```

Now you can execute the migrations console application like this:

```
./migrations
```

https://www.doctrine-project.org/projects/doctrine-migrations/en/3.0/index.html