Project Versions

latest ▼

Search

Go

Table Of Contents

6. Tools

26.1. Doctrine Console

The Doctrine Console is a Command Line Interface tool for simplifying common administration tasks during the development of a project that uses Doctrine 2.

Take a look at the **Installation and Configuration** chapter for more information how to setup the console command.

26.1.1. Display Help Information

Type **`php vendor/bin/doctrine`** on the command line and you should see an overview of the available commands or use the –help flag to get information on the available commands. If you want to know more about the use of generate entities for example, you can call:

```
$> php vendor/bin/doctrine orm:generate-entities --help
```

26.1.2. Configuration

Whenever the **`doctrine`** command line tool is invoked, it can access all Commands that wer registered by developer. There is no auto-detection mechanism at work. The Doctrine binar already registers all the commands that currently ship with Doctrine DBAL and ORM. If you want to use additional commands you have to register them yourself.

All the commands of the Doctrine Console require access to the **`EntityManager`** or **`DBAL`**Connection. You have to inject them into the console application using so called Helper-Sets. This requires either the **`db`** or the **`em`** helpers to be defined in order to work correctly.

Whenever you invoke the Doctrine binary the current folder is searched for a **`cli-config.php`** file. This file contains the project specific configuration:

```
<?php
$helperSet = new \Symfony\Component\Console\Helper\HelperSet(array(
    'db' => new \Doctrine\DBAL\Tools\Console\Helper\ConnectionHelper($conn)
));
$cli->setHelperSet($helperSet);
```

When dealing with the ORM package, the EntityManagerHelper is required:

```
<?php
$helperSet = new \Symfony\Component\Console\Helper\HelperSet(array(
    'em' => new \Doctrine\ORM\Tools\Console\Helper\EntityManagerHelper($em)
));
$cli->setHelperSet($helperSet);
```

The HelperSet instance has to be generated in a separate file (i.e. **`cli-config.php`**) that contains typical Doctrine bootstrap code and predefines the needed HelperSet attributes mentioned above. A sample **`cli-config.php`** file looks as follows:

```
<?php
// cli-config.php
require_once 'my_bootstrap.php';

// Any way to access the EntityManager from  your application
$em = GetMyEntityManager();

$helperSet = new \Symfony\Component\Console\Helper\HelperSet(array(
    'db' => new \Doctrine\DBAL\Tools\Console\Helper\ConnectionHelper($em->ge
nnection()),
    'em' => new \Doctrine\ORM\Tools\Console\Helper\EntityManagerHelper($em)
));
```

It is important to define a correct HelperSet that Doctrine binary script will ultimately use. The Doctrine Binary will automatically find the first instance of HelperSet in the global variable namespace and use this.

You have to adjust this snippet for your specific application or framework and us
their facilities to access the Doctrine EntityManager and Connection Resources.

26.1.3. Command Overview

The following Commands are currently available:

- **`help`** Displays help for a command (?)
- **`list`** Lists commands
- **`dbal:import`** Import SQL file(s) directly to Database.
- **`dbal:run-sql`** Executes arbitrary SQL directly from the command line.
- **`orm:clear-cache:metadata`** Clear all metadata cache of the various cache drivers.
- **`orm:clear-cache:query`** Clear all query cache of the various cache drivers.
- **`orm:clear-cache:result`** Clear result cache of the various cache drivers.
- **`orm:convert-d1-schema`** Converts Doctrine 1.X schema into a Doctrine 2.X schema.
- **`orm:convert-mapping`** Convert mapping information between supported formats.
- **`orm:ensure-production-settings`** Verify that Doctrine is properly configured for a produ environment.
- **`orm:generate-entities`** Generate entity classes and method stubs from your mapping information.
- **`orm:generate-proxies`** Generates proxy classes for entity classes.
- **`orm:generate-repositories`** Generate repository classes from your mapping informatio
- **`orm:run-dql`** Executes arbitrary DQL directly from the command line.
- **`orm:schema-tool:create`** Processes the schema and either create it directly on EntityManager Storage Connection or generate the SQL output.
- **`orm:schema-tool:drop`** Processes the schema and either drop the database schema of EntityManager Storage Connection or generate the SQL output.
- **`orm:schema-tool:update`** Processes the schema and either update the database schema EntityManager Storage Connection or generate the SQL output.

- **orm:convert:d1-schema** is alias for **orm:convert-d1-schema**.
- **orm:convert:mapping** is alias for **orm:convert-mapping**.
- **orm:generate:entities** is alias for **orm:generate-entities**.
- **orm:generate:proxies** is alias for **orm:generate-proxies**.
- **orm:generate:repositories** is alias for **orm:generate-repositories**.

Console also supports auto completion, for example, instead of **orm:clear-cache:query** you can use just **o:c:q**.

26.2. Database Schema Generation

SchemaTool can do harm to your database. It will drop or alter tables, indexes, sequences and such. Please use this tool with caution in development and not on a production server. It is meant for helping you develop your Database Schema, but NOT with migrating schema from A to B in production. A safe approach would be generating the SQL on development server and saving it into SQL Migration files that are executed manually on the production server.

SchemaTool assumes your Doctrine Project uses the given database on its own. Update and Drop commands will mess with other tables if they are not related to the current project that is using Doctrine. Please be careful!

To generate your database schema from your Doctrine mapping files you can use the**SchemaTool** class or the **schema-tool** Console Command.

When using the SchemaTool class directly, create your schema using the **createSchema()**method. First create an instance of the **SchemaTool** and pass it an instance of the**EntityManager** that you want to use to create the schema. This method receives an array of **ClassMetadataInfo** instances.

```
<?php
$tool = new \Doctrine\ORM\Tools\SchemaTool($em);
$classes = array(
  $em->getClassMetadata('Entities\User'),
$tool->createSchema($classes);
```

To drop the schema you can use the **dropSchema()** method.

```
<?php
$tool->dropSchema($classes);
```

This drops all the tables that are currently used by your metadata model. When you are changing your metadata a lot during development you might want to drop the complete database instead of only the tables of the current model to clean up with orphaned tables.

```
<?php
$tool->dropSchema($classes, \Doctrine\ORM\Tools\SchemaTool::DROP_DATABASE);
```

You can also use database introspection to update your schema easily with the**updateSchema()** method. It will compare your existing database schema to the passed array of **ClassMetdataInfo** instances.

```
<?php
$tool->updateSchema($classes);
```

If you want to use this functionality from the command line you can use the **schema-tool**command.

To create the schema use the **create** command:

```
$ php doctrine orm:schema-tool:create
```

To drop the schema use the **drop** command:

```
$ php doctrine orm:schema-tool:drop
```

If you want to drop and then recreate the schema then use both options:

```
$ php doctrine orm:schema-tool:drop
```

As you would think, if you want to update your schema use the **`update`** command:

```
$ php doctrine orm:schema-tool:update
```

All of the above commands also accept a **`--dump-sql`** option that will output the SQL for the ran operation.

```
$ php doctrine orm:schema-tool:create --dump-sql
```

Before using the orm:schema-tool commands, remember to configure your cli-config.php properly.

When using the Annotation Mapping Driver you have to either setup your autoloader in the cli-config.php correctly to find all the entities, or you can use the second argument of the **`EntityManagerHelper`** to specify all the paths of your entities (or mapping files), i.e. **`new\Doctrine\ORM\Tools\Console\Helper\EntityManagerHelper($em, $mappingPa`**

26.3. Entity Generation

Generate entity classes and method stubs from your mapping information.

```
$ php doctrine orm:generate-entities
$ php doctrine orm:generate-entities --update-entities
$ php doctrine orm:generate-entities --regenerate-entities
```

This command is not suited for constant usage. It is a little helper and does not support all the mapping edge cases very well. You still have to put work in your entities after using this command.

It is possible to use the EntityGenerator on code that you have already written. It will not b lost. The EntityGenerator will only append new code to your file and will not delete the old code. However this approach may still be prone to error and we suggest you use code repositories such as GIT or SVN to make backups of your code.

only and don’t put much additional logic on them. If you are however putting much more logic on the entities you should refrain from using the entity-generator and code your entities manually.

Even if you specified Inheritance options in your XML or YAML Mapping files the generator cannot generate the base and child classes for you correctly, because it doesn’t know which class is supposed to extend which. You have to adjust the entity code manually for inheritance to work!

26.4. Convert Mapping Information

Convert mapping information between supported formats.

This is an **execute one-time** command. It should not be necessary for you to call this method multiple times, especially when using the `--from-database` flag.

Converting an existing database schema into mapping files only solves about 70-80% of the necessary mapping information. Additionally the detection from an existing database canno detect inverse associations, inheritance types, entities with foreign keys as primary keys an many of the semantical operations on associations such as cascade.

There is no need to convert YAML or XML mapping files to annotations every tim you make changes. All mapping drivers are first class citizens in Doctrine 2 and can l used as runtime mapping for the ORM. See the docs on XML and YAML Mapping for a example how to register this metadata drivers as primary mapping source.

To convert some mapping information between the various supported formats you can use the `ClassMetadataExporter` to get exporter instances for the different formats:

```
<?php
$cme = new \Doctrine\ORM\Tools\Export\ClassMetadataExporter();
```

Once you have a instance you can use it to get an exporter. For example, the yml exporter:

```
<?php
$exporter = $cme->getExporter('yml', '/path/to/export/yml');
```

```
<?php
$classes = array(
  $em->getClassMetadata('Entities\User'),
  $em->getClassMetadata('Entities\Profile')
);
$exporter->setMetadata($classes);
$exporter->export();
```

This functionality is also available from the command line to convert your loaded mapping information to another format. The **`orm:convert-mapping`** command accepts two arguments, the type to convert to and the path to generate it:

```
$ php doctrine orm:convert-mapping xml /path/to/mapping-path-converted-to-xm
```

26.5. Reverse Engineering

You can use the **`DatabaseDriver`** to reverse engineer a database to an array of**`ClassMetadataInfo`** instances and generate YAML, XML, etc. from them.

Reverse Engineering is a **one-time** process that can get you started with a project. Converting an existing database schema into mapping files only detects abo 70-80% of the necessary mapping information. Additionally the detection from an existing database cannot detect inverse associations, inheritance types, entities with foreign keys as primary keys and many of the semantical operations on associations such as cascade.

First you need to retrieve the metadata instances with the **`DatabaseDriver`**:

```
<?php
$em->getConfiguration()->setMetadataDriverImpl(
    new \Doctrine\ORM\Mapping\Driver\DatabaseDriver(
        $em->getConnection()->getSchemaManager()
    )
);
```

```
$metadata = $cmf->getAllMetadata();
```

Now you can get an exporter instance and export the loaded metadata to yml:

```
<?php
$exporter = $cme->getExporter('yml', '/path/to/export/yml');
$exporter->setMetadata($metadata);
$exporter->export();
```

You can also reverse engineer a database using the **`orm:convert-mapping`** command:

```
$ php doctrine orm:convert-mapping --from-database yml /path/to/mapping-path
nverted-to-yml
```

Reverse Engineering is not always working perfectly depending on special cases. It will only detect Many-To-One relations (even if they are One-To-One) and will try t create entities from Many-To-Many tables. It also has problems with naming of foreig keys that have multiple column names. Any Reverse Engineered Database-Schema needs considerable manual work to become a useful domain model.

26.6. Runtime vs Development Mapping Validation

For performance reasons Doctrine 2 has to skip some of the necessary validation of metadata mappings. You have to execute this validation in your development workflow to verify the associations are correctly defined.

You can either use the Doctrine Command Line Tool:

```
doctrine orm:validate-schema
```

Or you can trigger the validation manually:

```
<?php
use Doctrine\ORM\Tools\SchemaValidator;
```

```
$errors = $validator->validateMapping();

if (count($errors) > 0) {
    // Lots of errors!
    echo implode("\n\n", $errors);
}
```

If the mapping is invalid the errors array contains a positive number of elements with error messages.

One mapping option that is not validated is the use of the referenced column name. It has to point to the equivalent primary key otherwise Doctrine will not work.

One common error is to use a backlash in front of the fully-qualified class-name Whenever a FQCN is represented inside a string (such as in your mapping definitions you have to drop the prefix backslash. PHP does this with **get_class()** or Reflection methods for backwards compatibility reasons.

26.7. Adding own commands

You can also add your own commands on-top of the Doctrine supported tools if you are using a manually built console script.

To include a new command on Doctrine Console, you need to do modify the **doctrine.php**fil a little:

```
<?php
// doctrine.php
use Symfony\Component\Console\Helper\Application;

// as before ...

// replace the ConsoleRunner::run() statement with:
$cli = new Application('Doctrine Command Line Interface', \Doctrine\ORM\Vers
::VERSION);
$cli->setCatchExceptions(true);
$cli->setHelperSet($helperSet);
```

```
// Register All Doctrine Commands
ConsoleRunner::addCommands($cli);

// Register your own command
$cli->addCommand(new \MyProject\Tools\Console\Commands\MyCustomCommand);

// Runs console application
$cli->run();
```

Additionally, include multiple commands (and overriding previously defined ones) is possibl
through the command:

```
<?php

$cli->addCommands(array(
    new \MyProject\Tools\Console\Commands\MyCustomCommand(),
    new \MyProject\Tools\Console\Commands\SomethingCommand(),
    new \MyProject\Tools\Console\Commands\AnotherCommand(),
    new \MyProject\Tools\Console\Commands\OneMoreCommand(),
));
```

26.8. Re-use console application

You are also able to retrieve and re-use the default console application. Just call **`ConsoleRunner::createApplication(...)`** with an appropriate HelperSet, like it is described in the configuration section.

```
<?php

// Retrieve default console application
$cli = ConsoleRunner::createApplication($helperSet);

// Runs console application
$cli->run();
```