CHAPTER 9

Command Line Tools

Doctrine 2 ships with powerful command line support. The command line tools can broadly be divided into those that support a database abstraction layer (DBAL-related operations) and those that provide object-relational mapping (ORM-related operations). A list of all commands available can be printed via the `list` command:

```
1   $ ./doctrine list
```

The `help` command, or the command parameter `--help` in conjunction with other commands, prints the instructions for a given command. For example, if we need help dealing with the `orm:info` command, we can ask for help like this:

```
1   $ ./doctrine orm:info --help
```

This will also work:

```
1   $ ./doctrine help orm:info
```

Both commands print the identical output to the console.

Setting Up the Command Line Tools

Before Doctrine 2 can be used on the command line, some basic configuration code is needed. Like the application itself, the command line tools require a ready-to-go database connection and entity manager if ORM-related commands are needed.

The configuration of the command line tools first looks somewhat strange. Doctrine 2 requires that a file called `cli-config.php` exists within the folder in which the tools are executed on the command line. If Doctrine 2 was installed using Composer, the command line tools are located in `vendor/bin`. Therefore, this is the place where a file called `cli-config.php` needs to be set up with the following basic code:

```
1   <?php
2   include '../../vendor/autoload.php';
3
4   use Doctrine\ORM\Tools\Setup;
```


M. Romer, *PHP Persistence*, DOI 10.1007/978-1-4842-2559-2_9

```
use Doctrine\ORM\EntityManager;

$paths = array(__DIR__ . "/../../entity/");
$isDevMode = true;

$dbParams = array(
        'driver' => 'pdo_mysql',
        'user' => 'root',
        'password' => '',
        'dbname' => 'app',
);

$config = Setup::createAnnotationMetadataConfiguration($paths,$isDevMode);
$em = EntityManager::create($dbParams, $config);

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

The code shown above is similar to the `index.php` file used in the demo app. First, autoloading is configured, then the entity manager is instantiated. If both clients, the web application and the command line tools, share the same credentials, externalizing this data may be helpful.

When the command line tools are invoked, Doctrine 2 automatically includes the `cli-config.php` file and also looks for a so-called *helper set* defined in the global namespace. The *helper set* provides the command line tools with the database connection via key `db` (needed for the DBAL commands) as well as the entity manager via key `em` (needed for the ORM commands).

DBAL Commands

Execute an SQL Statement

SQL commands can easily be executed via the command line with Doctrine's command line tooling. The command `dbal:run-sql` requires a single parameter, a valid SQL statement:

```
$ ./doctrine dbal:run-sql "SELECT * FROM users;"
```

The result of the query is printed to the console.

Import SQL Files

If you need to execute multiple statements, importing an SQL file via the `./doctrine dbal:import` command might be a great option. It takes one or more paths to SQL files, delimited by space:

```
1   $ ./doctrine dbal:import /tmp/import-data.sql
```

ORM Commands

Validate Persistence Configuration

One of the most helpful commands is `orm:validate-schema`, which validates the current persistence configuration and makes sure that it matches the existing data schema in the database. If all is good, the following command prints a positive message to the console:

```
1   $ ./doctrine orm:validate-schema
2   > [Mapping] OK - The mapping files are correct.
3   > [Database] OK - The database schema is in sync with the mapping files.
```

The command `orm:info` prints an overview of the application entities known to Doctrine:

```
1   $ ./doctrine orm:info
2   > Found 2 mapped entities:
3   > [OK] Entity\Post
4   > [OK] Entity\User
```

The Schema Tool

With the help of the commands

1. `orm:schema-tool:create`,
2. `orm:schema-tool:drop`
3. `orm:schema-tool:update`

one can manipulate a database data schema based on the entity persistence configuration. However, when running these commands, nothing actually happens—it's just a "dry run". If the commands are executed with parameter `--dump-sql`, again, only a dry run occurs. However, this time, the schema tool prints all SQL statements, which would otherwise have been fired against the database, to the screen. Only if one uses the parameter `--force`, does the schema tool finally execute the statements:

```
1   $ ./doctrine orm:schema-tool:drop --force
```

The command shown above makes all tables and data disappear:

```
1   > Dropping database schema...
2   > Database schema dropped successfully!
```

 **Danger! Potential loss of data!** The schema tool can delete tables and/or data. Use these commands with caution.

The execution of

```
1   $ ./doctrine orm:schema-tool:create
```

creates the data structure from scratch, while

```
1   $ ./doctrine orm:schema-tool:update
```

migrates an existing data schema from status quo to match the current persistence configuration, if the existing data schema is not yet up-to-date.

Generate Commands

With the help of the command `orm:generate-proxies`, the proxy classes for the entities defined can be created, which otherwise are created by Doctrine 2 automatically when needed. Via the command `orm:generate-entities`, the entity classes themselves can be generated; this is especially helpful if the persistence configuration is given via XML or YAML. Executing the command

```
1   $ ./doctrine orm:generate-entities /tmp
```

creates the entity classes in `/tmp`, or to be precise, in `/tmp/Entity`. More options for the command `orm:generate-entities` can be identified running the following command:

```
1   $ ./doctrine orm:generate-entities --help
```

With the help of the command `orm:generate-repositories`, repository classes can also be auto-generated.

Execute a DQL Command

Similar to `dbal:run-sql`, DQL statements can also easily be executed using the command `orm:run-dql`.

Cache-Related Commands

With the commands

1. `orm:clear-cache:metadata`,
2. `orm:clear-cache:query`
3. `orm:clear-cache:result`

the various Doctrine caches can be purged. These commands are especially helpful when deploying a new application version. They make sure that no outdated configuration or data is used in conjunction with a new application version—usually, this would lead to unrecoverable errors.

Converting Commands

Rarely used, but very helpful, are the commands `orm:convert-d1-schema` and `orm:convert-mapping`. The command `orm:convert-d1-schema` is used to transform the old Doctrine 1 persistence configuration format to the one used by Doctrine 2, which is very handy when upgrading an existing application. The command `orm:convert-mapping`, on the other hand, allows you to go from one Doctrine 2 mapping format to another, such as from XML to YAML.

Production-Ready Configuration

The following command validates that Doctrine's configuration is ready for production usage:

```
1   $ ./doctrine orm:ensure-production-settings
```

If this is the case, it prints a positive message on the console:

```
1   > Environment is correctly configured for production
```

Many configuration aspects can cause a negative result. This could be missing proxy classes created by running the proper command or missing caches. Both are considered by Doctrine 2 to be a requirement for usage in productive systems.

Custom commands Doctrine 2 allows you to develop custom commands that can be hooked into the command line tooling. More information about this topic can be found in the official documentation.[1]

[1] http://docs.doctrine-project.org/en/latest/reference/tools.html#adding-own-commands

Summary

The Doctrine CLI tools are extremely helpful. They can be used in shell or build scripts and also serve well when importing external data such as test fixtures or dealing with database schema changes. Since the CLI tools are also extendable, they are very versatile and should be part of every developer's toolkit.