SensioLabs

Doctrine

Table of contents

Introduction

Introduction

Doctrine allows you to abstract the storage into a database.

The usual Doctrine usage in Symfony consists in two layers:

- DBAL (DataBase Abstraction Layer)
- ORM (Object Relational Mapping)

Configuration

Open:

- .env
- config/packages/doctrine.yaml

Compatibility

Doctrine DBAL supports out of the box the following database systems:

- MySQL
- MariaDB
- Oracle
- Microsoft SQL Server
- PostgreSQL
- SQLite

Database layer

Doctrine DBAL uses the PDO API, and is automatically installed with doctrine/orm

You can still double-check your PDO extensions to make sure you have the proper driver for your database system.

```
$ symfony php --ri pdo
```

DBAL commands

Doctrine ships with multiple console commands to help you automate most actions.

You can start by using the `doctrine:schema:validate --skip-mapping` command to check the connection to your database.

The `doctrine:database:create` will create a database according to your configuration, while its counterpart `doctrine:database:drop` used with the `--force` option will delete it.

DBAL commands

```
# check first
$ symfony console doctrine:schema:validate --skip-mapping

$ symfony console doctrine:database:create
$ symfony console doctrine:database:drop --force
```

Exercise

1. Configure Doctrine to access the database. Let’s assume we will use a SQLite storage, located in var/data.db
2. Create the database with the console

Entities + Mapping

ORM

The ORM project serves as a link between your data classes and your database storage.

By means of system-agnostic entities, Doctrine will translate your data mapping to the storage and back again.

Entities

An Entity is the mapping configuration related to a PHP class: This configuration is available in multiple formats:

- Attributes (recommended) or annotations (deprecated)
- Yaml
- XML
- PHP

Entities

Mapping example:

```php
namespace App\Entity;

use App\Repository\BookRepository;
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity(repositoryClass: BookRepository::class)]
class Book
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column(type: 'integer')]
    private int $id;

    #[ORM\Column(type: 'string', length: 255)]
    private string $title;

    #[ORM\Column(type: 'decimal', precision: 5, scale: 2)]
    private float $price;

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

New entity

You can create/update an entity by adding manually some mapping to a PHP class, or you can use the MakerBundle's make:entity command to create both at the same time.

```
$ symfony console make:entity Book
```

Migrations

Migrations

- Migrations allow for a safe upgrade/downgrade of table definitions (up or down methods)
- All your operations are stored in versioned files (located in the migrations/ folder by default)
- Migrations are used as a more robust replacement do the command doctrine:schema:update

More here: **https://www.doctrine-project.org/projects/doctrine-migrations/en/current/index.html**

Check your status

- The previously seen command `doctrine:schema:validate` can be used to check the database connection and if the database schema is in sync with the mapping of your entities.
- You can also use the command `doctrine:mapping:info` to check the mapping information for each entity
- The `doctrine:migrations:status` command will give you informations on all the migrations of your system and their status

Check your status

```
# Check connection and db sync
$ symfony console doctrine:schema:validate

# Check your mapped entities
$ symfony console doctrine: mapping: info

# Overview about migrations
$ symfony console doctrine:migrations:statu
```

Generate a new migration

When you make changes to your entities like adding or removing properties or changing their types, create a new migration to apply the changes to your database.

This can be done with the MakerBundle `make:migration` command, which is a shortcut for the `doctrine:migrations:diff` command.

Generate a new migration

```
$ symfony console make:migration

# or
$ symfony console doctrine:migrations:diff

# Check the file content.
# If not satisfied, gently remove the file instead of executing it
```

Apply migrations

To apply the changes in your migration file to the database, run the `doctrine:migrations:migrate` command. This command will run every migration that's not yet been run. To run a specific migration, either its up or down method, use the `doctrine:migrations:execute` command with the `--up` or `--down` flag and the migration's Fully Qualified Class Name

Apply migrations

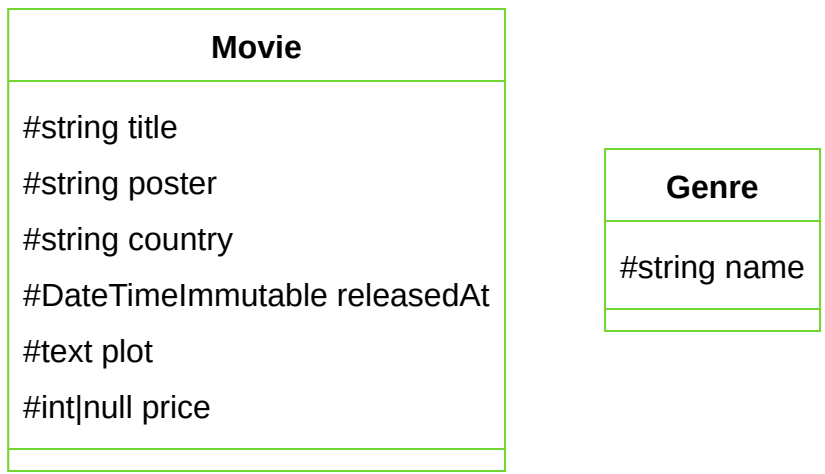
```
$ symfony console doctrine:migrations:migrate

# If something's wrong, rollback the last migration with
# its fully qualified classname
$ symfony console doctrine:migrations:execute {FQCN} --down
```

Exercise

1. Create new entities:
   - Movie
   - Genre
2. Generate a new migration.
3. Store both tables in your database.

Exercise


Movie
#string title
#string poster
#string country
#DateTimeImmutable releasedAt
#text plot
#int|null price
Genre
#string name