Doctrine and ORM 1

To interact with the database, Symfony uses the **Doctrine** library. This is what's called an ORM (*Object Relational Mapper*). Its role is to link the database with the objects we create in our code and to automate the queries.

For example, if we want to create a table named "article", we will create a class "Article", and Doctrine will read this class and create the corresponding table.

Symfony version 7 comes with a "**compose.yaml**" file to generate a PostgreSQL database container.
We can choose to use it or use any other DBMS.

If we choose to use the one provided with Docker, there's nothing more to do and we can move on to the next chapter titled "**Entity**".
Otherwise, we need to continue:

First, we need to tell Symfony where our DB is located. For that, go to the "**.env**" file.
(*Reminder: here we're working locally with no secret information. So sending this to git is not a problem, but when working with real passwords and a real database, create a ".env.local" file instead, which won't be sent to git.*)

Find the Doctrine-related lines, comment out those that don't match your DB and fill in the one that does:

```
// Structure:
DATABASE_URL="driver://username:password@url:port/dbName?
serverVersion=serverVersion&charset=utf8mb4"

# Example for MySQL/MariaDB
DATABASE_URL="mysql://root:root@127.0.0.1:3302/blog-symfony?serverVersion=11.2.2-
MariaDB&charset=utf8mb4"

# Example for PostgreSQL
DATABASE_URL="postgresql://app:root@127.0.0.1:5432/app?
serverVersion=16&charset=utf8"

# Don’t forget to uncomment the line that corresponds to your DB and fill in the
credentials
```

WARNING:
In the **php.ini** file used by your computer/server, the PDO extension for your driver must be enabled:

```
; ";" indicates a comment, meaning the line is not active
extension=pdo_pgsql
;extension=pdo_mysql
;extension=pdo_sqlite
```

Once that's set, type in the terminal:

```
symfony console doctrine:database:create
# Or shorter
symfony console d:d:c
```

Symfony will read the configuration defined in the ".env" or via Docker and create the database.

Entity

The first step to using Doctrine will be to create this first class. Let's say we want a table that contains messages. We could create the class manually, but Symfony will of course help us here.

We've seen "make:controller", now it's time for:

```
symfony console make:entity
symfony console make:entity EntityName
```

Symfony will then ask us several things:

- A name for the entity — here we'll choose **Message** (always start with a capital letter like this).
- It then asks if we want to enable real-time updates using Symfony UX Turbo. In brackets it shows `[no]`, which is the default if we hit Enter without typing anything — and that's what we'll do.
- Symfony may ask a few more questions, but we'll come back to them later. Just press **Enter** to finish.

A quick note on **UX Turbo**:
Since recent versions, Symfony has been pushing this library by default.
Previously, it had to be activated manually.
Now it's enabled automatically.

It allows you to simulate a SPA (Single Page Application) without writing JavaScript.
If we had chosen to enable the previous option, it would have allowed server-side data updates to be sent directly to all clients.
Not very useful in our example, but perfect for a chat, collaborative todo list or dynamic projects, as it allows users to see changes in real time without reloading.

We'll see that two files were created — let's ignore the repository for now and look at the entity:

- As expected, it starts with a namespace.
- Then a `use` statement for the corresponding repository.
- And `use` statements for Doctrine ORM.
- Above the class, an attribute links it to the repository.
- Inside the class we find by default an `id` property and its getter. This `id` has three Doctrine attributes: marks it as an ID, auto-generated, and a column.
  We also notice it's typed as `int|null`.

Now that we've seen this, let's rerun the same command as before:
make:entity, and again enter *Message*.
Symfony will detect that the entity already exists and offer to add new fields:

- Let's add "**content**"
- It asks for the property type — typing "?" shows all available types.
  By default (shown in square brackets), if we don't enter anything, it uses **string** (equivalent to "varchar").
  But here we'll choose **text**.
- The next questions may vary depending on the type — for a **string**, it asks for length (default is **255**).
- Then it asks whether the field can be null. Default is "**no**", which is fine.
- We're back to the beginning — Symfony asks if we want to add another field or stop.
- Add a "**createdAt**" field — Symfony recognizes the **"At"** in the name and suggests **datetime_immutable**. Nice. Accept with not null.
- Then add "**editedAt**", but change **datetime_immutable** to a standard **datetime**, and allow null (the difference is that datetime_immutable cannot be changed).
- Hit Enter again to finish.

Now look at the class:

- All our fields have been added.
- They are typed and constrained.
- Each one comes with a setter and a getter.

The next step is to push this class to the database.

Migrations

Now that we've created our entity, we just need to generate the **migration** — this is a file that contains all the SQL queries needed to turn our entities into tables.

If you look at your terminal, Symfony already told you what to do:

"Next: When you're ready, create a migration with symfony.exe console make:migration"

Note: depending on your setup, it might suggest `php bin/console`, which is equivalent to `symfony console`.
It shows `symfony.exe` here because it doesn't know you've added Symfony to your PATH.

```
symfony console make:migration
```

Symfony compares the current database state to your files and creates a migration reflecting the detected changes.
It asks us to review the file and then run another command.

Let's follow its instructions. In the migration file, you'll find 3 methods:

- `getDescription`, which does what it says.

- `up`, containing the SQL queries to update the DB (these depend on your DBMS — MySQL, PostgreSQL, etc.).
- `down`, which can reverse the migration if needed.
- You'll also notice a second table **messenger_messages** is created — it's used by a Symfony library, but we'll ignore it for now.
- Behind the scenes, a third table is also created — it tracks which migrations have been run.

If everything looks good, or after any modifications, push the migration using the next command:

```
symfony console doctrine:migrations:migrate
```

It warns that changes will be made to the database and that if we weren't careful, data could be lost — but we'll continue and confirm.

If you inspect your DB now, you'll see the new table.
Also, a table tracking applied migrations is present — this ensures migrations aren't reapplied every time.

To view your databases:

```
-- MySQL/MariaDB
SHOW DATABASES;

-- PostgreSQL
SELECT * FROM pg_catalog.pg_database;
```

To view tables:

```
-- MySQL/MariaDB
USE myDatabase; -- default is "app"
SHOW TABLES;

-- PostgreSQL
SELECT *
FROM pg_catalog.pg_tables
WHERE schemaname != 'pg_catalog' AND
    schemaname != 'information_schema';
```

If you rerun the command now, it will say there's nothing to change.

Bonus commands

Check detected entities and any errors:

```
symfony console doctrine:mapping:info
```

Check DB-PHP synchronization:

```
symfony console doctrine:schema:validate
```

Show SQL queries needed to update DB, without executing them:

```
symfony console doctrine:schema:update --dump-sql
```

Revert to the previous migration:

```
symfony console doctrine:migrations:migrate prev
```

⚠ Warning: Symfony is not perfect and might make mistakes in the `down()` method. Always review migrations — whether `up` or `down` — before applying them.