# Doctrine と ORM 1

データベースとやり取りするために、Symfony は **Doctrine** ライブラリを使用します。これは **ORM**（オブジェクト・リレーショナル・マッパー）と呼ばれるものです。
その役割は、データベースとコード内で作成したオブジェクトを関連付け、クエリの自動化を行うことです。

たとえば、「article」という名前のテーブルを作成したい場合、「Article」というクラスを作成します。すると Doctrine がこのクラスを読み取り、対応するテーブルを作成します。

Symfony 7 には PostgreSQL データベースコンテナを生成するための "**compose.yaml**" ファイルが付属しています。
それを使ってもいいですし、別の DBMS を使ってもかまいません。

もし Docker に付属のものを使うのであれば、これ以上設定する必要はなく、次の "**Entity**" の章に進んでかまいません。
別のものを使う場合は、以下を続けてください：

まず Symfony にデータベースの場所を教える必要があります。そのためには "**.env**" ファイルを編集します。
（補足：ローカルで作業しており秘密情報がないので git に送っても問題ありません。ただし実際のパスワードや本番用の DB を使う場合は代わりに ".env.local" を作成しましょう。これは git に送信されません。）

Doctrine に関連する行を見つけ、自分の DB に合わないものをコメントアウトし、合うものを記入してください：

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

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

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

# 自分の DB に合わせた行のコメントアウトを外し、認証情報を入力します。
```

**⚠注意:**
コンピュータ／サーバで使用されている **php.ini** ファイルにおいて、使用しているドライバ用の PDO 拡張が有効化されている必要があります：

```
;「;」はコメントアウトであり、無効な行です
extension=pdo_pgsql
```

```
;extension=pdo_mysql
;extension=pdo_sqlite
```

これが完了したら、ターミナルで以下を入力します：

```
symfony console doctrine:database:create
# または短縮形
symfony console d:d:c
```

Symfony は ".env" または Docker の構成を読み取り、データベースを作成します。

## エンティティ (Entity)

Doctrine を使う最初のステップは、エンティティクラスを作成することです。
たとえば、メッセージを含むテーブルを作成したいとします。クラスを手動で作成しても良いですが、Symfony が支援してくれます。

"make:controller" を以前に見ましたが、今度は以下のコマンドを使います：

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

Symfony はいくつかの質問をしてきます：

- エンティティの名前 — ここでは **Message** を選びます（最初は必ず大文字で始めます）
- 次に Symfony UX Turbo を有効にするか聞いてきます。`[no]` と表示されていれば Enter キーでそのまま進めば OK です。
- 他にもいくつか質問されることがありますが、ここでは Enter を押してスキップしてください。

**UX Turbo に関しての補足：**
最近のバージョンでは Symfony はこのライブラリをデフォルトで有効にしています（以前は手動で有効化する必要がありました）。
これは JavaScript を書かずに SPA（Single Page Application）のような体験を提供します。
もし前の質問で有効にしていれば、サーバ側のデータ更新をすべてのクライアントにリアルタイムで送信できるようになります。
今回はそこまで必要ないですが、チャットや共同編集などリアルタイム性の高いアプリケーションには最適です。

ここで 2 つのファイルが作成されているのがわかります。リポジトリは無視して、エンティティを見てみましょう：

- 予想通り namespace で始まっています。
- 対応するリポジトリの `use` 文があります。
- Doctrine ORM のアトリビュートも `use` で読み込まれています。
- クラス上部にはリポジトリへの関連を示すアトリビュートがあります。

- クラスの中には `id` プロパティとその getter が存在します。
  この `id` には 3 つの Doctrine アトリビュートがあります：ID であること、自動生成されること、カラムであること。
  また、型は `int|null` です。

これを確認したら、先ほどと同じ `make:entity` コマンドを再度実行します：
Message を再入力すると、既存のエンティティが見つかり、フィールドを追加するよう促されます。

- "**content**" を追加します。
- プロパティの型を聞かれます。`?` を入力するとすべての型が表示されます。
  デフォルトでは `[string]` と表示され、何も入力しなければそれが使われます。
  ここでは **text** を選びましょう。
- 型によってはさらに質問されます（たとえば string の場合は長さなど）
- `nullable` かどうか（null を許容するか）を聞かれます。ここでは **no**（デフォルト）で OK。
- もう一度質問のループに戻るので、新しいフィールドを追加できます。
- "**createdAt**" を追加します。名前に "At" があるため、Symfony は自動で **datetime_immutable** を提案します。これで OK。
- "**editedAt**" を追加し、型を **datetime** に変えて `nullable` を **yes** にします（immutable は変更不可）。

Enter を押して終了します。

クラスを確認すると：

- すべてのプロパティが追加されています。
- 型と制約が正しく設定されています。
- それぞれに setter と getter が生成されています。

次のステップは、このクラスをデータベースに反映させることです。

## マイグレーション (Migrations)

エンティティを作成したら、**マイグレーション** を生成します。これはエンティティを DB に反映するための SQL クエリが書かれたファイルです。

ターミナルを見ると、Symfony が次にやることを教えてくれています：

「次: マイグレーションを作成するには symfony.exe console make:migration を実行してください」

環境によっては `php bin/console` を使うように案内されるかもしれませんが、`symfony console` と同じです。
`symfony.exe` は Windows の PATH が通っていないためです。

```
symfony console make:migration
```

Symfony は現在の DB 状態とエンティティを比較し、変更点をマイグレーションとして出力します。
ファイルを確認するように指示されるので、それに従いましょう。

マイグレーションファイルには 3 つのメソッドがあります：

- `getDescription`：説明用
- `up`：変更を適用するための SQL
- `down`：変更を元に戻すための SQL
- Symfony のシステムで使われる **messenger_messages** テーブルも一緒に作られるかもしれませんが、ここでは無視します。
- 実際にはもう 1 つ、マイグレーションの履歴を追跡するテーブルも作成されます。

問題なければ、以下のコマンドでマイグレーションを実行します：

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

データベースが変更されるという警告が表示され、確認を求められますが、進めて OK です。

DB を確認すれば、テーブルが作成されたことが分かるでしょう。
また、どのマイグレーションが適用されたかを追跡するテーブルもあります。
これにより、毎回同じマイグレーションが実行されることはありません。

データベース一覧の表示：

```
-- MySQL/MariaDB
SHOW DATABASES;

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

テーブル一覧の表示：

```
-- MySQL/MariaDB
USE myDatabase; -- デフォルトは "app"
SHOW TABLES;

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

今再度マイグレーションを実行しても、変更点がないと言われるはずです。

## ボーナスコマンド

エンティティと検出結果の確認：

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

DB と PHP の整合性を確認：

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

SQL の確認（実行せず表示だけ）：

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

前のマイグレーションに戻す：

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

⚠ 注意：Symfony の `down()` は必ずしも正しく動作するとは限りません。
`up()` / `down()` の両方を実行前に必ず確認しましょう。