Créer la base de données de l’application

Table des matières

I. Contexte

Durée : 60 minutes.

Environnement de travail :

- Ordinateur avec Windows, macOS ou Linux
- Éditeur de code (PHPStorm ou Visual Studio Code sont conseillés)
- Un navigateur récent : Firefox, Chrome, Safari, Opera
- Si possible PHP > 7.2.5 installé avec les librairies/extensions suivantes :
    - Iconv
    - JSON
    - PCRE
    - Session
    - Tokenizer
- Composer
- Symfony CLI
- Un projet Symfony installé

Prérequis : connaître le langage PHP, SQL, avoir des notions informatiques pour utiliser un terminal.

Objectifs

- Introduire le concept de Model et de base de données relationnelle
- Connaître les différents types de données SQL
- Comprendre l'utilité d'un ORM
- Mettre en place la structure d'une base de données et de ses tables

Contexte

Dans ce cours, nous allons débuter notre projet fil rouge à partir d'un cahier des charges, contenant les besoins du client.

Également, nous allons nous intéresser à la création et à la structuration d'une base de données relationnelle à l'aide d'un ORM.

Mais, pour rappel, c'est quoi une base de données relationnelle ?

Une base de données relationnelle est un type de modèle qui enregistre ses données dans des tables, liées les unes aux autres par des relations.

Chaque table est composée de ligne et de colonnes.

Chaque ligne représente un enregistrement individuel de données identifié par un attribut unique appelé clé primaire.

Ces données sont donc organisées et cela rend leur accès flexible et efficace.

SQL quant à lui est un langage de requête structuré qui permet d'interagir avec ces bases en récupérant, enregistrant, manipulant ou supprimant des données.

De nos jours, beaucoup d'applications utilisent SQL.

On pense par exemple à l'extension MySQLI ou PDO pour PHP.

Mais avec un framework objet tel que Symfony ou Laravel, cela peut vite devenir compliqué.

C'est pourquoi nous allons voir l'utilisation d'un ORM comme Doctrine pour créer et structurer notre base de données relationnelles dans notre projet fil rouge.

II. Aborder un nouveau projet

A. Comprendre et modéliser le brief client

Dans le cadre de notre projet fil rouge, nous allons devoir créer une base de données, mais surtout la structurer et remplir celle-ci.

D'accord ? Mais que met-on dedans ? Quel est le besoin initial ?

Nous allons voir cela avec l'aide d'un cahier des charges !

Définition

Un cahier des charges pour le développement d'un logiciel ou d'un site web est un document détaillé qui définit des exigences, des fonctionnalités et des contraintes de projet.

Il sert de référence commune entre le client et l'équipe de développement pour s'assurer que toutes les parties comprennent les attentes et les objectifs du projet.

Le cahier des charges de notre projet est présent ici :

[cf. CDC Quai Antique.pdf]

En partant de ce cahier des charges, nous allons devoir modéliser et créer la base de données de notre site. Si les besoins ne sont pas clairs, la base de données ne sera pas adaptée à notre projet.

C'est pourquoi il faut toujours prendre du temps pour modéliser celle-ci pour éviter de mal stocker et structurer nos futures données. Cette étape est cruciale, elle impactera directement la couche applicative.

Pour cela, des méthodes de modélisation existent comme le langage UML ou Merise.

B. Introduction à la méthode Merise

Définition

La méthode Merise est une méthode de conception de systèmes d'information.

Elle débute par une phase d'analyse de l'existant et des besoins ; ici *via* notre cahier des charges.

Elle se décompose en 3 étapes : la modélisation conceptuelle MCD, la modélisation organisationnelle MLD et la modélisation physique MPD.

Ces modélisations facilitent la compréhension et la communication entre les différents acteurs du projet.

- Le Modèle Conceptuel de Données (MCD) est un schéma généraliste de votre base de données.
- Le Modèle Logique de Données (MLD) est une représentation sous forme de tableaux de la base de données.
- Le Modèle Physique de Données (MPD) décrit la façon dont la base de données est implémentée sur un Système de Gestion de Base de Données Relationnelles (SGBDR).

Méthode

Pour la construction d'un MCD, il faut comprendre les propriétés suivantes :

1. Les **entités** : une entité représente un objet, une personne, un concept ou un lieu qui peut être décrit et stocké dans une base de données.
2. Les **propriétés :** il faut pour chaque entité déterminer quelles en sont les propriétés. Un attribut représente une propriété de l'entité, telle que son nom, son âge, son adresse, etc.
3. Les **relations** : ensuite, vous devrez représenter les relations entre les entités. Une relation représente une association entre deux entités, telle qu'un client qui passe une commande. Pour cela, on utilise un losange avec un verbe à l'intérieur, représentant le type de relation/action entre les différentes entités.
4. Les **cardinalités** : enfin, vous allez déterminer la cardinalité, qui représente le nombre d'occurrences d'une entité qui peuvent être associées à une occurrence d'une autre entité dans une relation, par des signes du type **0, 1, 1..*, 0..**. La cardinalité est représentée par des chiffres situés près des extrémités des relations.

Voyons donc ensemble comment partir d'un cahier des charges pour modéliser notre MCD *(voire MPD)* et la base de notre projet.

III. Introduction à Doctrine

A. Introduction

Maintenant que nous savons quoi créer, il faut passer à la pratique, et rien de mieux qu'un ORM !

Un ORM *(Object-Relational Mapping)* est une technique permettant de faire le lien direct entre les données d'une base de données à des objets en programmation.

Son but est de simplifier l'accès aux données, au lieu d'utiliser le langage SQL, nous utiliserons une interface orientée objet.

Par conséquent, nous ne travaillerons plus avec des requêtes SQL traditionnelles, mais plutôt avec des objets à la place des tables et des attributs plutôt que des colonnes.

Cette couche d'abstraction nous permet de faire toutes les requêtes possibles sans écrire de SQL.

Bon nombre de frameworks utilisent des ORM, notamment Symfony ou encore Laravel.

Nous allons voir comment installer et maîtriser cette nouvelle technologie.

B. Installer Doctrine

L'ORM que nous allons installer se nomme Doctrine.

C'est l'ORM open source le plus répandu en PHP, permettant aux développeurs de travailler avec de l'objet plutôt que du SQL.

L'ORM Doctrine permet plusieurs choses :

- Créer une correspondance entre les classes PHP appelées « *Entité* » et les tables de la base de données.
- Requêter avec un langage propre à Doctrine, appelé DQL *(Doctrine Query Langage)* qui ressemble aux requêtes SQL, mais introduit la notion d'orientée objet.
- Gérer par lui-même les relations *(One-To-One, Many-To-One / One-To-Many, Many-To-Many)* et les jointures.
- Gérer par lui-même les opérations en cascades ou les transactions.
- Palier aux injections SQL.
- Et pleins d'autres choses encore : Doctrine[1].

1 https://www.doctrine-project.org/

Un exemple de l'utilisation de Doctrine ressemblerait à ceci :

```php
<?php

// On récupère l'Entity Manager avec notre configuration SQL (type de connexion),
// L'Entity Manager est l'objet de Doctrine qui nous permet d'interagir avec notre base de données.
$entityManager = new EntityManager($connection, $config);

// On crée un simple objet (ici Product) que l'on veut sauvegarder en base.
$product = new Product() ;

// On hydrate celui-ci de données.
$product->name = 'Casque audio sans fil';
$product->price = 50;

// Puis on sauvegarde avec Doctrine l'objet en base de données.
$entityManager->persist($product);
$entityManager->flush();

echo "Created Product with ID " . $product->getId() . "\n";

?>
```

Méthode **Installer Doctrine ORM Pack et MakerBundle**

À ce stade, vous devriez déjà avoir installé un projet Symfony et disposer d'une configuration minimale pour le faire tourner :

```
symfony check:requirements
```

Pour rappel, si ce n'est pas le cas, vous pouvez toujours créer un projet Symfony avec la commande :

```
symfony new my_project_directory --version="5.4.*"
```

Nous avons donc : un cahier des charges, une modélisation de notre base de données et un projet Symfony. Il nous faut maintenant créer notre base de données.

Deux possibilités existent : soit créer notre base en premier pour la mapper en code *(ce qu'on appelle Database First)* soit créer notre code en premier et le transformer en base *(ce qu'on appelle code first)*.

L'utilisation de Doctrine permet les deux, mais est plus généralement utilisée en code first.

C'est pourquoi nous allons installer notre composant, le configurer et créer nos entités par la suite.

Pour installer Doctrine, il vous faudra importer *via* Composer, l'ORM Symfony Pack qui contient toutes les librairies nécessaires de Doctrine propres à Symfony :

```
composer require symfony/orm-pack
```

Nous installerons également le composant MakerBundle qui permet aux développeurs de générer du code facilement en ligne de commande.

Cela nous aidera notamment à créer nos entités Doctrine *(nos tables)*, et plus encore !

```
composer require --dev symfony/maker-bundle
```

Une fois ces deux composants installés, vous pouvez vous assurer que votre projet est à jour avec la commande suivante :

```
composer update
```

Maintenant que votre projet est à jour et que vos composants sont fraîchement installés dans le dossier « *vendor/* », il ne nous reste plus qu'à configurer la connexion entre votre projet et votre base de données SQL.

C. Configurer Doctrine avec le fichier .env

Méthode **Qu'est-ce que le .env ?**

En Symfony, le fichier « *.env* » est le fichier de configuration global de votre projet se trouvant à la racine de votre dossier.

Il permet aux développeurs de stocker des variables d'environnement.

Ces variables sont souvent utilisées pour paramétrer des connexions à la base de données, à des API, pour contenir des clés secrètes ou encore des valeurs qui sont propres à votre logique métier.

C'est avec ce fichier que nous allons mettre en place une connexion entre Doctrine de Symfony et votre base de données MySQL.

Mais avant ça, pensez à récupérer vos identifiants SQL, car nous en aurons besoin *(souvent avec l'utilisateur root, même si ce n'est pas recommandé).*

IV. Créer notre première table

A. Créer une entité Doctrine

Définition

Une entité est une classe PHP qui représente un objet du domaine métier.

La classe est utilisée pour interagir avec une base de données relationnelle.

Le mapping automatique de Doctrine est une fonctionnalité qui permet de générer automatiquement les classes d'entités à partir d'une base de données relationnelle et vice versa.

Vous pouvez créer vos entités de deux manières :

- Manuellement
- *Via* le composant Symfony/maker-bundle

Vous pouvez tout à fait personnaliser le mapping automatique qui aura été créé par le MakerBundle en amont, en utilisant les annotations pour définir des contraintes de validation, des relations entre les entités, des héritages, etc.

Pour résumer, si nous voulons une table dans notre base de données appelée par exemple « *Product* » avec un nom de produit et un prix, nous aurons une entité équivalente en PHP à :

```php
<?php

namespace App\Entity;

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

/** @ORM\Entity(repositoryClass=ProductRepository::class) */
class Product
{
   /** @ORM\Column(type="string", length=32, unique=true) */
    private string $name;

    /** @ORM\Column(type="smallint", options={"unsigned": true}) */
    private int $price = 0;

    [assesseurs…]
}
```

À savoir : chaque entité dispose d'assesseurs, notamment des getters et des setters. Ce sont des méthodes qui permettent d'accéder aux propriétés privées de votre entité *(de votre classe)* pour ajouter ou récupérer les valeurs.

Fondamental

Pour résumer, une entité Symfony est composée :

- D'une classe PHP, qui représente l'entité,
- Des annotations Doctrine, qui sont utilisées pour le mappage entre la classe PHP et la table correspondante dans votre base de données,
- D'attributs de la classe liés à ces annotations,
- Des méthodes d'accès *(getters et setters)* utilisées pour lire et écrire dans les propriétés de l'entité.

Complément

L'ensemble des types définis dans les annotations Doctrine sont différents de ceux de PHP et de ceux de la base de données, même s'ils sont mappés aux deux. En voici quelques-uns :

- **String** : il permet de mapper une propriété de type string à une colonne de type VARCHAR dans la base de données,
- **Integer** : il permet de mapper une propriété de type integer à une colonne de type INT dans la base de données,
- **Float** : il permet de mapper une propriété de type float à une colonne de type Float dans la base de données,
- **Decimal** : il permet de mapper une propriété de type décimal à une colonne de type DÉCIMAL dans la base de données,
- **Boolean** : il permet de mapper une propriété de type booléen à une colonne de type BOOLEAN.

Nous avons donc pour notre entité « *Product* » un attribut « *$name* » de type string en PHP, string en Doctrine, mais équivalent à VARCHAR en SQL.

Pareil pour « *$price* », qui est un integer en PHP, un smallint en Doctrine et un SMALLINT en SQL.

Maintenant que vous savez ce qu'est une entité, il suffit de la créer *via* le MakerBundle et sa commande « `make:entity` ».

Nous allons débuter notre projet de restaurant et par conséquent créer cette première entité.

Mais juste avant il nous faut notre base de données !

C'est elle qui recevra nos futures entités donc après avoir configuré le fichier « *.env.local* » avec vos identifiants de base de données, assurez-vous que celle-ci soit bien créée avec la commande :

```
1 php bin/console doctrine:database :create
```

Créons maintenant notre entité « *Restaurant* » :

```
1 php bin/console make:entity Restaurant
```

Pensez à mettre des majuscules aux noms des entités, ce sont des classes.

Elles respectent donc les PSR PHP et la convention de nommage « *PascalCase* ».

Cette commande va créer une classe d'entité dans le répertoire de projet « *src/Entity* » et ajoutera par défaut un id *(clé primaire)* dans sa table/entité.

À la suite, le MakerBundle va vous poser une série de questions pour configurer les propriétés de votre entité. Il va vous demander :

- Son nom
- Ses propriétés
- Ses relations avec d'autres entités
- Ses validations

Il nous faut donc créer par question : un nom, une description, les heures d'ouvertures, le nombre d'invités maximum, la date de création de l'entité et la date de sa mise à jour.

Une fois que vous avez répondu à toutes les questions et créé votre entité comme désigné dans le MPD, il suffit d'appuyer sur votre touche « *Entrer* » et MakerBundle va générer le code PHP correspondant à votre entité dans le dossier « *src/Entity* » et sa classe de repository dans le dossier « *src/Repository* ».

Définition

Mais qu'est-ce qu'un repository ?

Un repository est une classe allant de pair avec son entité.

Elle est une couche d'abstraction vous permettant d'effectuer des requêtes sur votre entité et par conséquent, sur les tables enregistrées en base de données.

C'est avec cette classe, que vous pourrez par la suite, récupérer vos Restaurants par critère depuis votre base ou effectuer d'autres opérations personnalisées.

Il est important de savoir que pour chaque entité, Doctrine ORM met à disposition un repository par défaut.

Ce repository par défaut peut être utilisé aisément à travers tous les contrôleurs dans vos applications pour interagir avec votre base de données, en allant le chercher depuis « *src/Repository* ».

Exemple

```php
<?php

use App\Entity\Restaurant;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\Persistence\ManagerRegistry;

class RestaurantRepository extends ServiceEntityRepository
{
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Restaurant::class);
    }

    public function findActiveRestaurants() : array
    {
        return $this->createQueryBuilder('r')
            ->where('r.isActive = true')
            ->getQuery()
            ->getResult();
    }
}
```

B. Migrer votre entité en base

Désormais, nous allons devoir mapper pour la première fois une entité en base.

Pour ce faire nous devons utiliser un composant nous permettant de migrer nos entités avec Doctrine. Celui-ci se nomme « *DoctrineMigrationsBundle* », il fait partie du « *symfony/orm-pack* » que nous avons installé en amont.

Même si ce n'est pas nécessaire, voici la commande :

```
composer require doctrine/doctrine-migrations-bundle "^3.0"
```

Avec la commande suivante, vous allez créer une classe de Migration permettant à Symfony de transformer votre entité en un ensemble de requêtes SQL ayant pour but de créer votre table en base de données.

```
php bin/console make:migration
```

Une fois la classe créée dans le dossier de « *migrations/* » il ne nous reste plus qu'à exécuter les requêtes SQL de cette classe *via* la commande suivante :

```
php bin/console doctrine:migrations:migrate
```

Si vous allez dans votre base de données avec MySQL, PHPMyAdmin ou autre, vous verrez votre nouvelle table fraîchement créée par Symfony, sans avoir tapé une seule ligne de requête SQL !

V. Créer et lier nos autres entités

A. Lier vos entités avec des relations

Pour l'instant, Doctrine vous permet de créer des entités, de les mapper et d'interagir simplement avec vos données. À ce stade, vous êtes en mesure de créer une table indépendante « *Restaurant* », et pourquoi pas une table « *Picture* ». Mais comment lier mes tables entre elles ? Comment peut-on jouer avec nos données ? Par exemple, admettons qu'un utilisateur souhaite sortir tous les produits d'une certaine catégorie ?

C'est ce qu'on appelle une « *relation* ». Et pour cela, il est crucial de comprendre les 3 types de relations possibles entre les entités. Le type de relation mis en place est relativement important, car il impactera directement la performance de l'application et la façon dont le code fonctionnel sera écrit.

B. La relation 1-1, One-To-One

Définition

Une relation 1 pour 1 fait référence à une association entre deux entités où chaque enregistrement dans la première entité est lié à un seul enregistrement dans la deuxième entité, et vice versa.

Mais comment traduire cela en SQL ? Eh bien, une des entités *(l'entité « propriétaire »)* contiendra une clé étrangère qui fera référence à l'entité associée.

```php
<?php

namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

class Personne
{
    /**
     * Un objet Personne ne peut être lié qu'à un objet ADN unique.
     * Si je crée un autre objet Personne avec le même ADN (même id) il y a aura une erreur
     * @ORM\OneToOne (targetEntity="App\Entity\ADN")
     * @ORM\JoinColumn (name="adn_id", referencedColumnName="id")
     */
    private $adn;
```

```
16
17      //…
18  }
19
20  class ADN
21  {
22      //...
23  }
```

C. La relation 1-n / n-1, One-To-Many / Many-To-One

Définition

Une relation 1 pour plusieurs *(aussi plusieurs pour 1)* est la relation la plus courante.

Elle fait référence à une association entre deux entités où chaque enregistrement dans la première entité peut être associé à plusieurs enregistrements d'une autre entité.

Et en SQL ?

L'entité du côté Many contiendra toujours la clé étrangère qui fera référence à l'entité associée.

Par exemple : nous avons une entité « *Ecole* » qui peut avoir plusieurs « *Eleve* ».

Alors nous avons Many « *Eleve* » to One « *Ecole* ». La clé étrangère sera donc du côté « *Eleve* ».

```
1   < ? php
2
3   namespace App\Entity;
4
5   use Doctrine\ORM\Mapping as ORM ;
6
7   class Eleve
8   {
9       /**
10       * @ORM\Id
11       * @ORM\GeneratedValue
12       * @ORM\Column(type="integer")
13       */
14      private $id;
15
16      /**
17       * @ORM\Column(type="string")
18       */
19      private $nom;
20
21      /**
22       * @ORM\ManyToOne(targetEntity="Ecole", inversedBy="eleves")
23       * @ORM\JoinColumn(name="ecole_id", referencedColumnName="id")
24       */
25      private $ecole;
26
27      // ...
28  }
```

D. La relation n-n / Many-To-Many

À l'inverse de la Many To One, cette relation Many To Many nous sert si des enregistrements font références à plusieurs autres enregistrements.

Reprenons le cas précédent des objets « *Ecole* » et « *Eleve* ».

Si nos « *Eleve* » seraient en même temps dans une autre « *Ecole* » *(admettons qu'ils sont dans une école nommée A certains jours et dans une autre école nommée B d'autres jours)*, il nous faudra retenir en SQL qu'un « *Eleve* » peut-être dans l'école A, mais également dans l'école B. Et que chaque « *Ecole* » puisse avoir plusieurs élèves.

Pour cela, Doctrine va créer une table de jointure, nommée « *ecole_eleve* » qui contiendra les colonnes suivantes :

ecole_id : cette colonne stocke l'identifiant de l'école pour chaque lien entre une école et un élève. Elle est une clé étrangère qui fait référence à la clé primaire de la table école.

eleve_id : cette colonne stocke l'identifiant de l'élève pour chaque lien entre une école et un élève. Elle est une clé étrangère qui fait référence à la clé primaire de la table élève.

Chaque ligne dans cette table de jointure représente une association entre une « *Ecole* » et un « Eleve ». Donc, si un « *Eleve* » appartient à plusieurs « *Ecole* », il y aura plusieurs lignes pour cet « *Eleve* » dans la table de jointure.

Il est important de noter que Doctrine gère automatiquement la création et la maintenance de cette table de jointure pour vous.

À comprendre également, que ce n'est pas parce qu'une table supplémentaire de jointure a été ajoutée par Doctrine, qu'une entité en plus sera créée.

VI. L'essentiel

Pour enregistrer des données sur votre projet, vous n'avez plus besoin de passer par SQL.

Vous êtes désormais en mesure de créer vos tables et leurs relations directement depuis l'ORM Doctrine et Symfony.

Pensez donc à bien modéliser votre base en amont, à créer vos entités avec leurs propriétés, à lier celles-ci et à les mapper au sein de votre base avec les classes de migration !

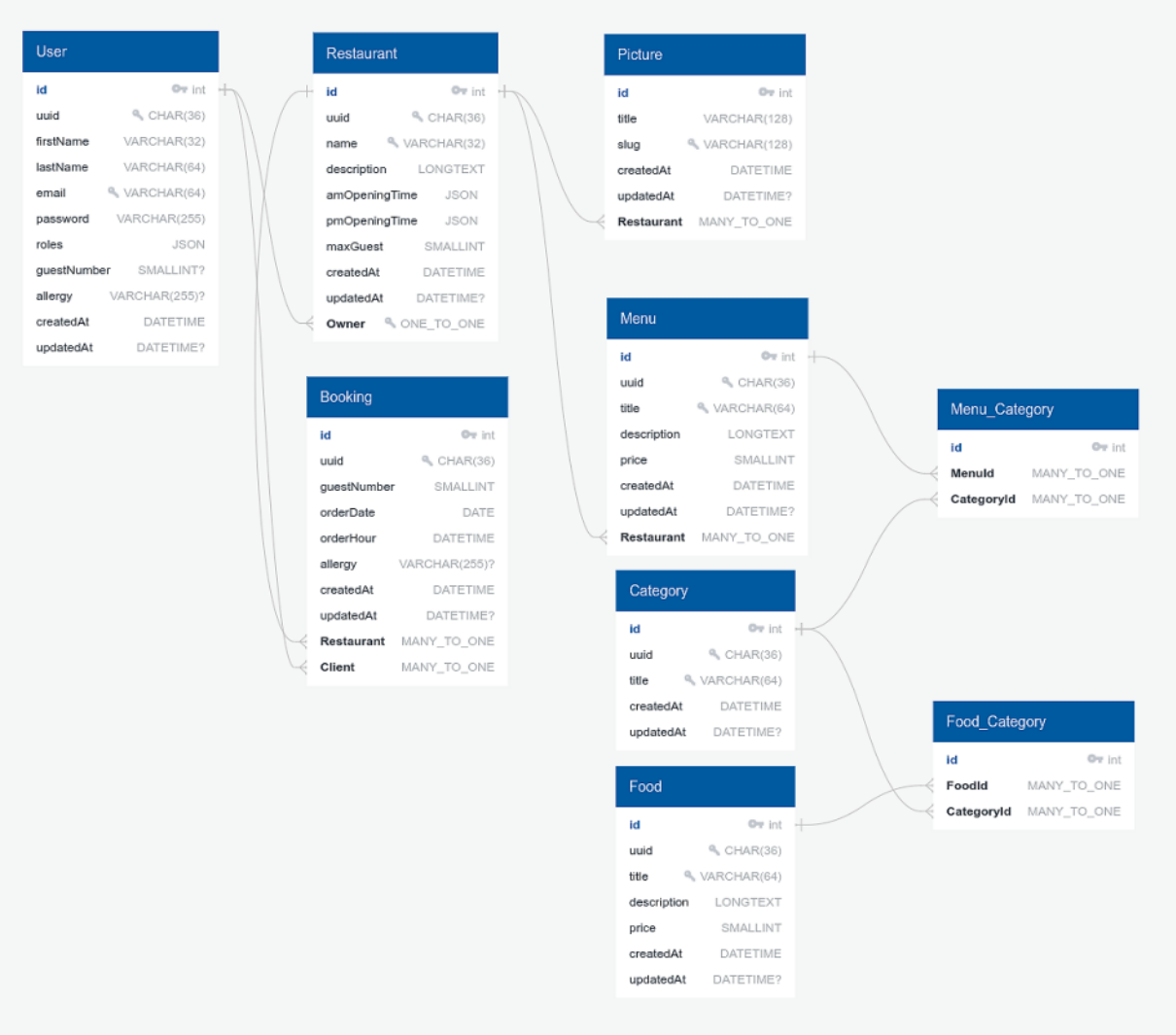
On pourra s'aider du récapitulatif de commandes suivant :

```
1 php bin/console doctrine:database:create    // Pour créer une base de données depuis le .env
2 php bin/console make:entity Restaurant    // Pour créer une entité Restaurant avec ses attributs
3 php bin/console make:migration    // Pour créer la migration contenant les requêtes SQL de la classe
4 php bin/console doctrine:migrations:migrate    // Pour exécuter les requêtes SQL de la migration
```

VII. Pour aller plus loin

A. Pour aller plus loin

Maintenant que vous savez créer une entité et la lier à une autre, il ne reste plus qu'à finir de créer vos entités et structurer votre base de données suivant le MPD suivant :



Vous devez donc, *via* Makerbundle :

- Créer l'entité « *Booking* » avec ses attributs,
- Créer L'entité « *Menu* » avec ses attributs,
- Créer L'entité « Category » avec ses attributs,
- Créer L'entité « *Food* » avec ses attributs,
- Ajouter vos relations ManyToMany entre « *Category* » et « *Food* », ainsi que « *Menu* » et « *Category* », ce qui créera les deux tables de jointes,
- Ajouter votre relation Many « *Booking* » To One « *Restaurant* ».

Quant à l'entité « *User* », **nous la créerons plus tard**, dans un contexte de sécurité.