Doctrine et ORM 1

Pour discuter avec la base de donnée, Symfony utilise la bibliothèque **Doctrine**. Cette dernière est ce qu'on appelle un ORM (*Object Relation Mapper*). Son rôle est de faire le lien entre la base de donnée et les objets que l'on crée dans notre code et automatise les requêtes.

Pour prendre un exemple, si on souhaite créer une table "article" alors on va créer une classe "article" et Doctrine lira cette classe et créera la table correspondante.

Symfony arrive en version "7" avec un fichier "**compose.yaml**" pour générer un gestionnaire de base de donné postgresql. Nous pouvons choisir de l'utiliser ou d'utiliser toute autre BDD.

Si nous choisissons d'utiliser celle fourni avec docker, nous n'avons plus rien à faire de plus et nous pouvons passer au chapitre suivant nommé "**Entity**". Sinon il faudra continuer :

Avant tout, on a besoin d'indiquer à symfony où se trouve notre BDD, pour cela rendons nous dans le fichier "**.env**".
(*Pour rappel, ici on travail sur local avec aucune information secrète. Donc envoyer cela sur git n'est pas un problème, mais lorsque vous travaillez avec de vrai mot de passes et une vrai BDD, créez un fichier ".**env.local**" qui lui ne sera pas envoyé sur git.*)

Trouvez les lignes correspondantes à doctrine, commentez celles qui ne correspondent pas à votre bdd et remplissez celle qui correspond :

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

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

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

# N’oubliez pas de décommenter la ligne correspondant à votre base de données et
de remplir les identifiants
```

ATTENTION : Dans le fichier **php.ini** utilisé par votre ordinateur/serveur, vous devez avoir l'extension pdo correspondante à votre driver d'activé:

```
; ";" indique un commentaire et donc que la ligne n'est pas activée
extension=pdo_pgsql
;extension=pdo_mysql
;extension=pdo_sqlite
```

Une fois cela validé, vous pouvez taper dans le terminal :

```
symfony console doctrine:database:create
# Ou en résumé
symfony console d:d:c
```

Symfony va lire la configuration définie dans le ".env" ou par docker et créer la BDD.

Entity

La première étape pour faire usage de Doctrine, cela va être de créer cette première classe, imaginons qu'on souhaite une table contenant des messages. Et bien on pourrait créer la classe à la main, mais évidement, Symfony va nous mâcher le travail.

On avait vu le "make:controller", maintenant c'est au tour de :

```
symfony console make:entity
symfony console make:entity nomEntité
```

Symfony va ensuite nous demander plusieurs choses :

- Un nom pour l'entité, ici choisissons **Message** (essayez de toujours mettre une majuscule comme dans l'exemple).
- Il nous demande si on souhaite ajouter la capacité de diffuser les mises à jours de l'entité en utilisant Symfony UX Turbo. Entre crochet il est indiqué `[no]`, c'est la réponse par défaut si on appui sur entrée sans rien mettre, et c'est ce que l'on va faire. Symfony nous pose d'autres questions auxquels on pourrait répondre, mais on y reviendra plus tard. appuyons sur **entrée** pour mettre fin à cela.

Pour revenir un instant sur **UX Turbo**, depuis les dernières version de Symfony, ils ont poussé l'usage de cette bibliothèque. Avant il fallait l'activer volontairement pour pouvoir l'utiliser. Maintenant elle est activé par défaut.

Elle permet d'imiter une SPA sans avoir à écrire de javascript. Si on avait choisi d'activer l'option précédente, Cela aurait permis d'envoyer les informations mises à jour côté serveur directement aux différents clients. Pour notre exemple ce n'est pas très utile, mais si on fait un tchat, une todoList colaborative ou autre projet dynamique, cela permettrait de voir les changements apportés par un utilisateur directement sans avoir à recharger.

On voit qu'il a créé deux fichiers, laissons pour l'instant de côté le repository et regardons l'entité.

- Cela commence évidement par un namespace.
- Puis le use du repository qui lui correspond.
- Et le use de doctrine en tant que ORM.
- Au dessus de la classe, on trouve un attribut indiquant la liaison de la classe avec le repository du même nom.
- Dans notre classe nous trouvons par défaut une propriété id et son getter. Cet id se voit attribué 3 lignes indiquant que c'est un ID, qu'il doit être généré automatiquement et que c'est une colonne.

On remarquera aussi que cet id est de type int ou null.

Maintenant qu'on a vu cela, retapons exactement la même commande que précédemment, le "**make:entity**" et indiquons lui encore une fois *Message*.
Symfony va voir que cette entité existe déjà et va nous proposer de lui ajouter des propriétés :

- Ajoutons lui "**content**"
- Il va nous demander le type de cette propriété, en tapant "?" vous pourrez voir tous les types disponible. Mais par défaut, on le voit entre crochet, si on ne lui indique rien, il prendra le type **string** qui équivaut à "varchar", nous allons plutôt choisir **text**.
- Les questions suivantes peuvent varier selon le type choisi, par exemple sur un type **string**, il nous demande la taille de notre string, par défaut **255**.
- Ensuite il nous demande si ce champ peut être null avec par défaut "**no**", parfait, je ne le souhaite pas null.
- Nous revoilà à la case départ, il nous redemande si on veut ajouter une autre propriété ou alors s'arrêter.
- Ajoutons un champ "**createdAt**"... Mais que voyons nous? il nous propose par défaut un champ "datetime_immutable". parfois symfony reconnaît certains noms de propriété et propose le type adapté (*Ici il reconnait le mot "At"*). Validons cela avec un not null.
- Enfin ajoutons un "**editedAt**", mais changeons ce "datetime_immutable" par un "datetime" classique. et indiquons lui qu'il peut être null (la difference étant que datetime_immutable n'a pas le droit d'être changé).
- Nous pouvons à nouveau appuyer sur entrer pour terminer la modification.

Retournons voir notre classe :

- Toutes nos propriétés ont été ajouté.
- Elles sont typé et conditionné.
- Elles sont toute accompagné d'un setter et d'un getter.

La prochaine étape va être d'envoyer cette classe en BDD.

Migrations

Maintenant que nous avons créé notre entité, il ne nous reste plus qu'à créer la "**migration**", c'est un fichier qui va contenir toute les requêtes SQL pour convertir les entités en tables.

Pour cela, si vous regardez votre terminal, symfony vous a déjà indiqué quoi faire :

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

à noter que selon l'installation, il peut aussi indiquer "php bin/console" qui est la même chose que "symfony console". Il nous indique "symfony.exe" car il ne sait pas que nous avons ajouté le fichier symfony.exe à nos path.

```
symfony console make:migration
```

Symfony a comparé la BDD actuelle avec ses fichiers et a créé une migration qui correspond aux changements remarqués.
Il nous demande maintenant de vérifier le fichier puis de lancer une nouvelle commande.

Obéissons lui. On va trouver dans la migration 3 méthodes :

- getDescription, qui fait ce qu'elle dit.
- up qui contient les requêtes SQL à lancer pour mettre à jour notre BDD (celles-ci sont générées pour le SGBDD choisie, les requêtes varierons donc selon si on a choisi MySQL, PostgreSQL ou autre).
- down qui permettra d'inverser cette migration si jamais on a un problème.
- On notera que cette première migration crée aussi une seconde table **messenger_messages** liée à une bibliothèque de Symfony, mais on ne s'en occupera pas maintenant.
- De façon cachée, il va même créer une troisième table qui lui servira de guide pour savoir quelles sont les migrations qui ont été effectuées ou non.

Si cela nous convient, ou après avoir modifié des éléments, on peut envoyer la migration comme il nous l'indique après notre précédente commande :

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

il nous indique que l'on va changer notre BDD et que si on n'a pas fait attention, cela pourrait nous faire perdre des données, mais on souhaite continuer alors validons cela.

Si on regarde notre BDD, on verra notre nouvelle table définie. Ainsi qu'une table qui sauvegarde quels sont les migrations que l'on a déjà faite. Pour qu'il ne relance pas toutes les migrations à chaque fois.

Voir les bases de données :

```
-- MySQL/MariaDB
SHOW DATABASES;
-- PostgreSQL
SELECT * FROM pg_catalog.pg_database;
```

Voir les tables :

```
-- MySQL/MariaDB
USE maDatabase; -- par défaut "app"
SHOW TABLES;
-- PostgreSQL
SELECT *
FROM pg_catalog.pg_tables
WHERE schemaname != 'pg_catalog' AND
    schemaname != 'information_schema';
```

Si je lui relance la commande maintenant, il me dira qu'il n'y a rien à changer.

Commandes bonus

Vérifier quelles sont les entités détectées et si il n'y pas d'erreur :

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

Vérifier la synchronisation entre PHP et la BDD:

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

Affiche les requêtes SQL necessaire pour mettre à jour la BDD mais ne les executes pas.

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

Revenir à la migration précédente.

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

Attention pour cette dernière, symfony n'est pas parfait et peut se tromper sur le down, vérifier la migration avant de la lancer est toujours une bonne pratique, que ce soit avec un "up" ou un "down".