

SYMFONY

Base de données et ORM
Anthony PARIS - Formateur

QU'EST CE QU'UN ORM?

Symfony utilise Doctrine pour la gestion de la base de données. Doctrine est un ORM (Object Relational Mapping) qui va nous permettre d'écrire et lire dans notre base de données en utilisant que du PHP, pas de requête SQL donc (à moins que ce soit un cas vraiment précis - raw sql).

Disons par exemple nous avons notre objet \$article qui à été créé et nous devons l'enregistrer en base de données, habituellement on ferait une requête INSERT, avec un ORM, nous n'allons pas du tout nous fatiguer avec tout cela, nous réaliserons un \$orm->save(\$article) et c'est bon. Plus de SQL dans du PHP.

Doctrine ORM implémente 2 patterns objets pour mapper un objet PHP à des éléments d'un système de persistance :

Le pattern "Data Mapper".

Le pattern "Unit of Work".

PATTERN : DATA MAPPER

Le Data Mapper est une couche qui synchronise la donnée stockée en base avec les objets PHP. En d'autres termes :

il peut insérer, mettre à jour des entrées en base de données à partir de données contenues dans les propriétés d'un objet ;

il peut supprimer des entrées en base de données si les "entités" liées sont identifiées pour être supprimées ;

il "hydrate" des objets en mémoire à partir d'informations contenues en base.

L'implémentation dans le projet Doctrine de ce Data Mapper s'appelle l'Entity Manager, les entités ne sont que de simples objets PHP mappés.

Le grand avantage d'utiliser un Data Mapper, c'est que les objets sont complètement indépendants du système de stockage.

Pour une raison de performance et d'intégrité, l'Entity Manager ne synchronise pas directement chaque changement avec la base de données

PATTERN : UNIT OF WORK

L'Unit of Work est lui utilisé pour gérer l'état des différents objets hydratés par l'Entity Manager. La synchronisation en base ne s'effectue que quand on exécute la méthode "flush" et est effectuée sous forme d'une transaction qui est annulée en cas d'échec.

L'Entity Manager fait donc le lien entre les "Entités", qui sont de simples objets PHP, et la base de données :

à l'aide de la fonction find, il retrouve et hydrate un objet à partir d'informations retrouvées en base ;

à l'aide de la fonction persist, il ajoute l'objet manipulé dans l'Unit of Work ;

à l'aide de la fonction flush, tous les objets "marqués" pour être ajoutés, mis à jour et supprimés conduiront à l'exécution d'une transaction avec la base de données.

Pour marquer un objet pour la suppression, il faut utiliser la fonction remove.

<https://martinfowler.com/eaaCatalog/unitOfWork.html>

CREATION ENTITE

Nous allons commencer par créer notre entité Article.

- Une image pour illustrer l'article (picture: picture)
- Un titre (title: string)
- Un contenu, oui forcément (content: text)
- Une ou plusieurs catégories (categories: Category[])
- Une date de publication et de dernière modification (publicationDate: datetime, lastUpdateDate: datetime)
- Et un booléen pour savoir si l'article est publié (isPublished: boolean)

Nous avons donc 7 champs pour l'entité article en plus de l'id.

Le champ categories est un tableau de type Category qui est aussi une entité, ce qui veut dire que nous devrons créer une entité Category et réaliser une relation entre Category et Article.

Même chose pour l'image, on pourra utiliser cet «objet» pour différentes classes.

Nous verrons cela plus tard, pour l'instant, nous allons créer l'entité Article avec les champs title, content, publicationDate, lastUpdateDate et isPublished. Le champ id sera automatiquement généré.

CREATION ENTITE

Une entité est une classe PHP comme nous l'avons vu jusqu'à présent.
Disposant d'attribut, de getter et setter ainsi que de méthodes.

Elle disposera en plus d'annotations (commentaires) qui seront interprétées par doctrine pour intéragir avec votre base de données (relations, contraintes, valeurs, etc...).

Les entités sont listées dans le dossier src/entity il est possible de les générer à la main cependant nous allons utiliser le maker bundle précédemment installé pour générer rapidement nos entités.

```
D:\wamp64\www\sf4m2icours> php bin/console make:entity
```

Lancez la commande en étant à la racine de votre projet

CREATION ENTITE

Rentrez le nom de votre classe «Article».

Maintenant il faut renseigner les champs de notre entité:

- title: string (255), nullable = no
- content: text, nullable = yes
- publicationDate: datetime, nullable = yes
- lastUpdateDate: datetime, nullable = no
- isPublished: boolean, nullable = no

Appuyez deux fois sur entrer pour sortir de la commande , nous pouvons donc ouvrir le fichier src/Entity/Article.php

CREATION ENTITE

Dans src/entity/article.php

```
<?php

namespace App\Entity;

use App\Repository\ArticleRepository;
use Doctrine\DBAL\Types\Types;
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity(repositoryClass: ArticleRepository::class)]
class Article
{
    #[ORM\Id]
    #[ORM\GeneratedValue]
    #[ORM\Column]
    private ?int $id = null;

    #[ORM\Column(length: 255)]
    private ?string $title = null;

    #[ORM\Column(type: Types::TEXT, nullable: true)]
    private ?string $content = null;

    #[ORM\Column(type: Types::DATETIME_MUTABLE)]
    private ?\DateTimeInterface $publicationDate = null;

    #[ORM\Column]
    private ?bool $isPublished = null;

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

    public function getTitle(): ?string
    {
        return $this->title;
    }
}
```

ANNOTATION ENTITY

L'attribut Entity spécifie que la classe est considérée en tant qu'entité.

L'attribut le plus important est repositoryClass, qui permet de spécifier un repository spécifique pour l'entité (on y reviendra plus tard).

L'attribut Id identifie la clé primaire de la table.

L'attribut GeneratedValue délègue la responsabilité de l'unicité de l'id au système de persistance choisi.

<https://www.doctrine-project.org/projects/doctrine-orm/en/latest/reference/attributes-reference.html>

ANNOTATION ENTITY

L'attribut "Column"

L'attribut `##[ORM\Column]` permet de mapper une propriété PHP à une colonne de la base de données.

Par défaut, le nom de la colonne de la base de données sera le nom de la propriété PHP, mais il est possible de le surcharger. Si une propriété n'est pas marquée avec l'attribut, elle sera complètement ignorée.

Cet attribut a de nombreuses propriétés, toutes documentées. Voici les plus utiles :

- `unique` qui définit que la valeur doit être unique pour la colonne donnée ;
- `nullable` qui autorise/interdit la valeur nulle ;
- `length`, pour les chaînes de caractères, définit la longueur.
- `name`, si l'on souhaite un nom différent de celui dans la table

Ces attributs ne servent pas à la validation d'un objet, elles ne sont utilisées que pour le mapping et la génération de la base de données.

TYPE DE MAPPING

Les types définis dans les attributs Doctrine ne correspondent ni aux types en PHP ni à ceux de la base de données, mais sont mappés aux deux. En voici quelques-uns :

Type Doctrine	Type PHP	Type en base de données (MySQL)
string	string	VARCHAR
integer	integer	INT
boolean	boolean	BOOLEAN
date	\DateTime	DATETIME
datetime	\DateTime	TIMESTAMP
blob	stream resource	BLOB

<https://www.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/types.html#types>

BASE DE DONNÉES

Actuellement nous avons juste une classe PHP, il faut maintenant créer la table dans la base de données. Mais avant nous allons d'abord configurer l'accès à notre base de données.

Pour cela nous allons ouvrir le fichier .env qui se trouve à la racine du projet, sur la ligne 32 nous avons:

```
24 # MAILER_DSN=smtp://localhost
25 ###< symfony/mailer ###
26
27 ###> doctrine/doctrine-bundle ###
28 # Format described at https://www.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html#connection-options
29 # For an SQLite database, use: "sqlite:///%kernel.project_dir%/var/data.db"
30 # For a PostgreSQL database, use: "postgresql://db_user:db_password@127.0.0.1:5432/db_name?serverVersion=11&clientVersion=11"
31 # IMPORTANT: You MUST configure your server version, either here or in config/packages/doctrine.yaml
32 DATABASE_URL=mysql://db_user:db_password@127.0.0.1:3306/db_name?serverVersion=5.7
33 ###< doctrine/doctrine-bundle ###
34 |
```

BASE DE DONNÉES

Nous allons naturellement utiliser MySQL, mais il y a plusieurs intégrations possibles
SQLite, PostgreSQL, ...

Nous allons créer un fichier `.env.local` à la racine de notre projet et coller cette ligne dans ce fichier.

Alors pourquoi utiliser un nouveau fichier?

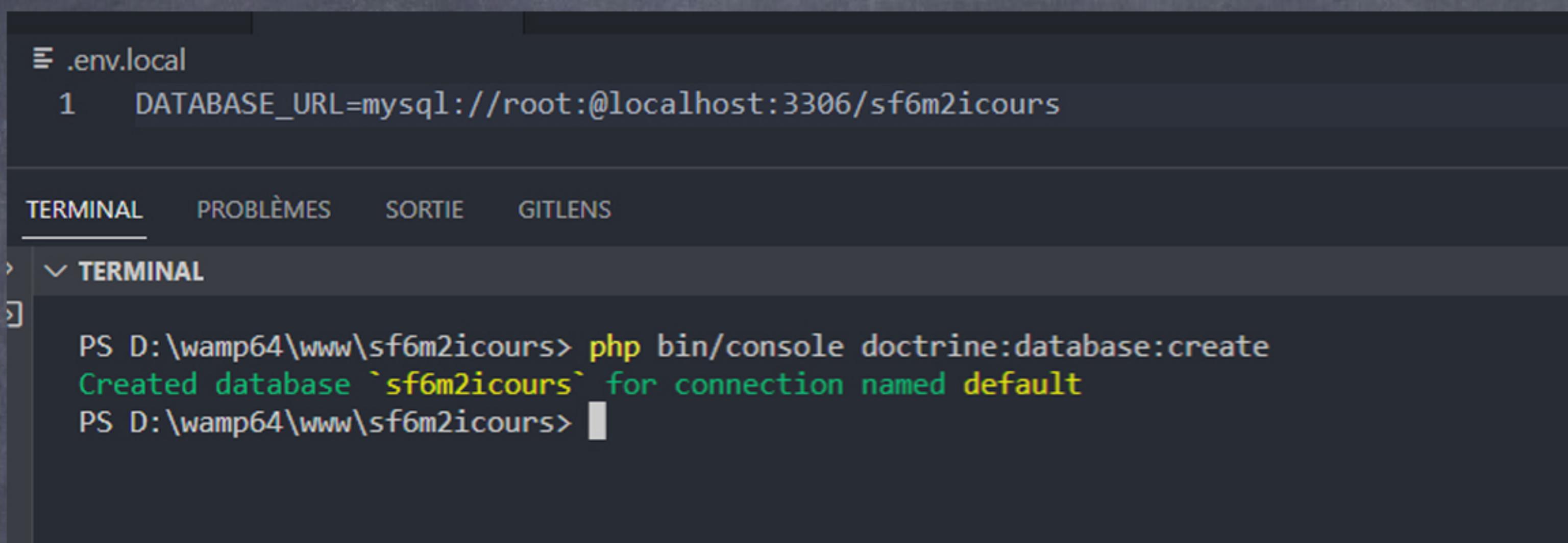
Pour la simple raison que le fichier `.env` n'est pas ignoré par git, ce qui veut dire que si vous publiez votre code sur Github, tout le monde aura accès à votre mot de passe.

Par contre le fichier `.env.local` est lui ignoré par git, il ne quittera jamais votre ordinateur.

```
#  
# * .env           contains default values for the environment vari  
# * .env.local     uncommitted file with local overrides  
# * .env.$APP_ENV   committed environment-specific defaults  
# * .env.$APP_ENV.local uncommitted environment-specific overrides  
#
```

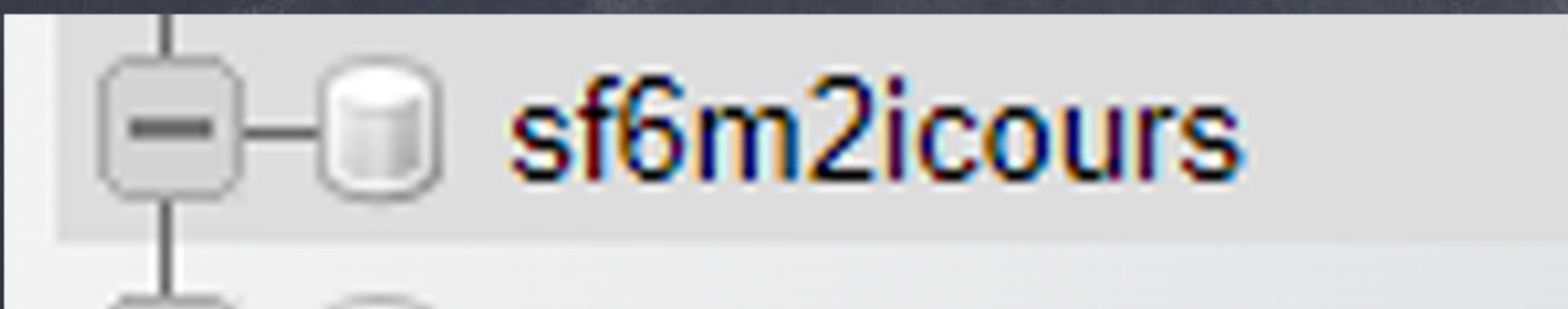
BASE DE DONNÉES

Créez un fichier à la racine env.local avec pour contenu votre base de données et lancez la commande de création de database :



The screenshot shows a terminal window within a code editor interface. The terminal tab is active, displaying the command: `PS D:\wamp64\www\sf6m2icours> php bin/console doctrine:database:create`. The output shows the database was created successfully: `Created database `sf6m2icours` for connection named default`. The terminal window has a dark background with light-colored text.

Dans phpmyadmin :



BASE DE DONNÉES

Actuellement sur la version 6 de symfony pour éviter d'avoir des requêtes superflues à cause du système de migrations il est possible de désactiver une option. Cela nous permet de ne plus voir des requêtes «étranges» dans le profiler de SF6

```
config > packages > ! doctrine_migrations.yaml
You, il y a 2 semaines | 1 author (You)
1 doctrine_migrations:
2   migrations_paths:
3     # namespace is arbitrary but should be different from App\Migrations
4     # as migrations classes should NOT be autoloaded
5     'DoctrineMigrations': '%kernel.project_dir%/migrations'
6   enable_profiler: '%kernel.debug%'
7
```

INFO PRODUCTION ET ENV

Chaque application est la combinaison de code et d'un ensemble de configurations qui dicte comment ce code doit fonctionner. La configuration peut définir la base de données utilisées, si quelque chose doit être mis en cache ou comment la journalisation doit être détaillée.

Dans Symfony, la notion d "environnements" est l'idée que la même base de code peut être exécutée en utilisant plusieurs configurations différentes. Par exemple, le dev environnement doit utiliser une configuration qui rend le développement facile et convivial, tandis que le prod environnement doit utiliser un ensemble de configurations optimisées pour la vitesse.

Une application typique Symfony commence par trois environnements: dev, prod et test. Comme mentionné, chaque environnement représente un moyen d'exécuter la même base de code avec une configuration différente. Il n'est donc pas surprenant que chaque environnement charge ses propres fichiers de configuration individuels. Ces différents fichiers sont organisés par environnement:

Dans le dossier config/packages vous aurez souvent un attribut de type :
when@XXXX qui définit le comportement en fonction de l'environnement dev/prod/test

INFO PRODUCTION ET ENV

Pour exécuter l'application dans chaque environnement, modifiez la APP_ENV variable d'environnement. Pendant le développement, cela se fait dans .env ou dans .env.local:

APP_ENV = dev|prod|test

Sachez qu'il est possible de créer ses propres «environnement»

La procédure est disponible ici :

<https://symfony.com/doc/current/configuration.html>

Essayez de changer l'environnement dans .env.local dev>prod

Pour supprimer l'erreur ajouter ceci dans la configuration de monolog
Et oui le channel deprecation est utilisé dans l'environnement de prod

```
monolog:  
    channels: ['services', 'controller', "deprecation"]
```

VARIABLES D'ENVIRONNEMENT EN PRODUCTION

En production, les .env fichiers sont également analysés et chargés à chaque demande. Par conséquent, la façon la plus simple de définir des vars env est de déployer un .env.local fichier sur vos serveurs de production avec vos valeurs de production.

Pour améliorer les performances, vous pouvez éventuellement exécuter la dump-env commande

Après avoir exécuté cette commande, Symfony chargera le .env.local.php fichier pour obtenir les variables d'environnement et ne passera pas de temps à analyser les .env fichiers

```
PS D:\wamp64\www\sf4m2icours> composer dump-env prod
```

CRÉATION DES TABLES

Revenons sur notre projet, nous pouvons maintenant créer la table article dans cette nouvelle base de données.

```
PS D:\wamp64\www\sf6m2icours> php bin/console doctrine:schema:update --dump-sql
CREATE TABLE article (id INT AUTO_INCREMENT NOT NULL, title VARCHAR(255) NOT NULL, content LONGTEXT DEFAULT NULL, publication_date DATETIME NOT NULL, is_pinned TINYINT(1) NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4 COLLATE `utf8mb4_unicode_ci` ENGINE = InnoDB;
CREATE TABLE messenger_messages (id BIGINT AUTO_INCREMENT NOT NULL, body LONGTEXT NOT NULL, headers LONGTEXT NOT NULL, queue_name VARCHAR(190) NOT NULL, created_at DATETIME NOT NULL, available_at DATETIME NOT NULL, delivered_at DATETIME DEFAULT NULL, INDEX IDX_75EA56E0FB7336F0 (queue_name), INDEX IDX_75EA56E016BA31DB (available_at), INDEX IDX_75EA56E016BA31DB (delivered_at), PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4 COLLATE `utf8mb4_unicode_ci` ENGINE = InnoDB;
```

Cette commande va nous afficher la requête SQL à exécuter.

Nous avons bien une requête CREATE TABLE.

On rappelle la même commande avec l'option --force pour lui dire «exécute cette requête SQL dans la base de données».

```
PS D:\wamp64\www\sf6m2icours> php bin/console doctrine:schema:update --force
Updating database schema...
2 queries were executed

[OK] Database schema updated successfully!
```

Remarquez la création d'une table messenger : Utilisez pour envoyer des notifications clients en mode asynchrone : <https://www.youtube.com/watch?v=OBWU-lIZIU4>

CRÉATION DES TABLES

Nous pouvons vérifier sur phpmyadmin :

The screenshot shows the phpMyAdmin interface with the following details:

- Success Message:** MySQL a retourné un résultat vide (c'est à dire aucune ligne). (traitement en 0,0009 seconde(s).)
- Query:** SELECT * FROM `article`
- Table Structure:** A table with columns: id, title, content, publication_date, last_update_date, is_published.
- Operations:** Opérations sur les résultats de la requête, with a link to "Créer une vue".

CRÉATION DES ENTITÉS RESTANTES

Nous avions également deux autres entity :

- image

Le but étant de dissocier «l'image» sous un objet afin de pouvoir le ré-utiliser par exemple dans des annonces ou des profils

- > chemin (text : no nullable)
- > alt (string : 255, no nullable)
- > isPublished (boolean : no nullable)
- > dateCreation (datetime : no nullable)

Ouvrez ensuite image.php et modifiez :

```
#[ORM\Column(type: 'datetime', options: ['default' => 'CURRENT_TIMESTAMP'])]
private ?\DateTimeInterface $dateCreation = null;
```

Cela permet de spécifier une valeur par défaut ici «current_timestamp»

CRÉATION DES ENTITÉS RESTANTES

IL faut ensuite lancer la commande d'ajout à la base de données :

```
PS D:\wamp64\www\sf4m2icours> php bin/console doctrine:schema:update --dump-sql
The following SQL statements will be executed:

CREATE TABLE image (id INT AUTO_INCREMENT NOT NULL, chemin LONGTEXT NOT NULL, alt VARCHAR(255) NOT NULL, is_published TI
NYINT(1) NOT NULL, date_creation DATETIME DEFAULT CURRENT_TIMESTAMP NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4
COLLATE `utf8mb4_unicode_ci` ENGINE = InnoDB;
PS D:\wamp64\www\sf4m2icours> php bin/console doctrine:schema:update --force

Updating database schema...
1 query was executed

[OK] Database schema updated successfully!
```

Il nous reste catégorie :

- category
- > label (string : 255 , no nullable)
- > contenu (text : yes nullable)
- > isPublished (boolean : no nullable)
- > dateCreation (datetime : no nullable)

CRÉATION DES ENTITÉS RESTANTES

On effectue la même modification sur Category.php

```
#[ORM\Column(type: Types::DATETIME_MUTABLE, options: ['default' => 'CURRENT_TIMESTAMP'])]
private ?\DateTimeInterface $dateCreation = null;
```

On lance ensuite l'ajout à la base de données :

```
PS D:\wamp64\www\sf4m2icours> php bin/console doctrine:schema:update --dump-sql
```

```
The following SQL statements will be executed:
```

```
CREATE TABLE category (id INT AUTO_INCREMENT NOT NULL, label VARCHAR(255) NOT NULL, contenu LONGTEXT DEFAULT NULL, is_published TINYINT(1) NOT NULL, date_creation DATETIME DEFAULT CURRENT_TIMESTAMP NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8mb4 COLLATE `utf8mb4_unicode_ci` ENGINE = InnoDB;
```

```
PS D:\wamp64\www\sf4m2icours> php bin/console doctrine:schema:update --force
```

```
Updating database schema...
```

```
1 query was executed
```

```
[OK] Database schema updated successfully!
```

LES RELATIONS

À un certain niveau de complexité, vos objets PHP vont interagir les uns avec les autres

- Dans une relation 1-1 : un objet A correspond à un objet B ;
- Dans une relation 1-n : un objet A est lié à de nombreuses instances de B ;
- Dans une relation n-n : des objets de type A ont de multiples relations avec des objets de type B.

Doctrine supporte différents types d'associations :

- One-To-One : 1 entité est liée à 1 entité ;
- Many-To-One : plusieurs entités liées à 1 entité (Liée à une One-To-Many) ;
- One-To-Many : une entité liée à plusieurs ;
- ManyToMany : plusieurs entités liées à plusieurs.

UNIDIRECTIONNALITE / BIDIRECTIONNALITE

Dans une relation il y a toujours une notion de propriétaire (Notion d'unidirectionnalité et de bidirectionnalité)

Cette notion est également simple à comprendre : une relation peut être à sens unique ou à double sens. On ne va traiter dans ce chapitre que les relations à sens unique, dites unidirectionnelles. Cela signifie que vous pourrez écrire `$entiteProprietaire->getEntiteInverse()`, mais pas `entiteInverse->getEntiteProprietaire()`.

Attention, cela ne nous empêchera pas de récupérer les infos dans l'autre sens on utilisera juste une autre méthode, via le repository.

Cette limitation nous simplifie la façon de définir les relations. Pour bien travailler avec, il suffit juste de se rappeler qu'on ne peut pas faire `$entiteInverse->getEntiteProprietaire()`.

Pour des cas spécifiques, ou des préférences dans votre code, cette limitation peut être contournée en utilisant les relations à double sens, dites bidirectionnelles.

RELATION ONE TO ONE

Pour établir une relation One-To-One entre deux entités Article et Image, il faut relancer la création de l'entité Announce et ajouter un attribut image pointant vers l'entity image :

```
PS D:\wamp64\www\sf6m2icours> php bin/console make:entity
```

```
Class name of the entity to create or update (e.g. VictoriousPopsicle):
```

```
> Article
```

```
Your entity already exists! So let's add some new fields!
```

```
New property name (press <return> to stop adding fields):
```

```
> image
```

```
Field type (enter ? to see all types) [string]:
```

```
> relation
```

```
What class should this entity be related to?:
```

```
> Image
```

```
What type of relationship is this?
```

Type	Description
ManyToOne	Each Article relates to (has) one Image. Each Image can relate to (can have) many Article objects
OneToMany	Each Article can relate to (can have) many Image objects. Each Image relates to (has) one Article
ManyToMany	Each Article can relate to (can have) many Image objects. Each Image can also relate to (can also have) many Article objects
OneToOne	Each Article relates to (has) exactly one Image. Each Image also relates to (has) exactly one Article.

```
Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:
```

```
> OneToOne
```

```
Is the Article.image property allowed to be null (nullable)? (yes/no) [yes]:
```

```
>
```

```
Is the Article.image property allowed to be null (nullable)? (yes/no) [yes]:
```

```
>
```

```
Do you want to add a new property to Image so that you can access/update Article objects from it - e.g. $image->getArt:
```

```
>
```

```
updated: src/Entity/Article.php
```

```
Add another property? Enter the property name (or press <return> to stop adding fields):
```

```
>
```

```
Success!
```

```
Next: When you're ready, create a migration with php bin/console make:migration
```

```
#[ORM\OneToOne(cascade: ['persist', 'remove'])]  
private ?Image $image = null;
```

RELATION ONE TO ONE

Explication :

Tout d'abord, j'ai choisi de définir l'entité article comme entité propriétaire de la relation, car un article « possède » une Image. On aura donc plus tendance à récupérer l'image à partir de l'article que l'inverse.

Cela permet également de rendre indépendante l'entité Image: elle pourra être utilisée par d'autres entités que Advert, de façon totalement invisible pour elle.

Ensuite, vous voyez que seule l'entité propriétaire a été modifiée, ici Article. C'est parce qu'on a une relation unidirectionnelle, rappelez-vous, on peut donc faire \$article->getImage(), mais pas \$image->getArticle(). Dans une relation unidirectionnelle, l'entité inverse, ici Image, ne sait en fait même pas qu'elle est liée à une autre entité, ce n'est pas son rôle.

RELATION ONE TO ONE

Par défaut, une relation est facultative, c'est-à-dire que vous pouvez avoir un Article qui n'a pas d'Image liée. C'est le comportement que nous voulons pour l'exemple : on se donne le droit d'ajouter une annonce sans forcément trouver une image d'illustration. Si vous souhaitez forcer la relation, il faut ajouter choisir dans les options que la relation est obligatoire :

```
Is the Article.image property allowed to be null (nullable)? (yes/no) [yes]: ici no  
>  
Do you want to add a new property to Image so that you can access/update Article objects from it - e.g. $image->getArticle()? (yes/no) [no]:  
>
```

Si vous souhaitez pouvoir accéder à l'article depuis l'image il faut choisir yes pour la deuxième option.

Par contre sachant que notre image peut être utilisée par une autre entité, il ne faut pas le faire dans notre cas.

RELATION ONE TO ONE

Opérations de cascade :

Imaginez que vous supprimiez une entité Article via un `$em->remove($article)`. Si vous ne précisez rien, Doctrine va supprimer l' article mais garder l'entité Image liée.

Si vos images ne sont liées qu'à des articles, alors la suppression de l'article doit entraîner la suppression de l'image, sinon vous aurez des Images orphelines dans votre base de données. C'est le but de cascade.

Attention, si vos images sont liées à des articles mais aussi à d'autres entités, alors vous ne voulez pas forcément supprimer directement l'image d'un annonce, car elle pourrait être liée à une autre entité.

On peut cascader des opérations de suppression, mais également de persistance.

Cependant, dans le cas d'entités liées, si on fait un `$em->persist($article)`, qu'est-ce que Doctrine doit faire pour l'entité Image contenue dans l'entité Article ?

Il ne le sait pas et c'est pourquoi il faut le lui dire : soit en faisant manuellement un `persist()` sur l'article et l'image, soit en définissant dans l'attribut de la relation qu'un `persist()` sur Article doit se « propager » sur l'Image liée.

RELATION ONE TO ONE

Dans notre cas actuel nous avons ceci :

```
# [ORM\OneToOne(cascade: ['persist', 'remove'])]
private ?Image $image = null;
```

Nous devons supprimer le remove en cascade

```
# [ORM\OneToOne(cascade: ['persist'])]
private ?Image $image = null;
```

RELATION ONE TO ONE

Launchons la commande pour actualiser notre entity et générer le setter + getter

```
PS D:\wamp64\www\sf4m2icours> php bin/console make:entity --regenerate  
  
This command will generate any missing methods (e.g. getters & setters) for a class or all classes in a namespace.  
  
To overwrite any existing methods, re-run this command with the --overwrite flag  
  
Enter a class or namespace to regenerate [App\Entity]:  
> App\Entity\Article  
  
updated: src/Entity/Article.php  
  
Success!
```

Nous réaliserons le même process pour l'entité «category.php»
-Ajout de \$image + attribut + commande regenerate

La création d'une relation many to one est assez similaire (le propriétaire sera toujours du côté «many»)

RELATION MANY TO MANY

Nous avons dans notre cas une relation ManyToMany entre Article et Category, il y a plusieurs Category (Many) lier à (To) un ou plusieurs Article (Many). L'entité propriétaire sera Article, car on utilisera plus l'article pour rechercher la catégorie et non l'inverse. Nous allons donc rajouter la relation dans la classe src/Entity/Article.php

Pas de cascade ici car je n'ajouterais pas un article en même temps qu'une catégorie.

```
PS D:\wamp64\www\sf6m2icours> php bin/console make:entity  
Class name of the entity to create or update (e.g. TinyChef):  
> Article  
  
Your entity already exists! So let's add some new fields!  
  
New property name (press <return> to stop adding fields):  
> Categories  
  
Field type (enter ? to see all types) [string]:  
> relation  
  
What class should this entity be related to?:  
> Category  
  
What type of relationship is this?
```

```
Relation type? [ManyToOne, OneToMany, ManyToMany, OneToOne]:  
> ManyToMany
```

```
Do you want to add a new property to Category so that you can access/update Article objects from it - e.g. $category->getArticles()? (yes/no) [yes]:  
>
```

A new property will also be added to the Category class so that you can access the related Article objects from it.

```
New field name inside Category [articles]:  
>
```

```
updated: src/Entity/Article.php  
updated: src/Entity/Category.php
```

Pas besoin de regénérer les entités car cela se fait automatiquement si vous passez par la méthode CLI ;)

article.php

category.php

```
#[ORM\ManyToMany(targetEntity: Category::class, inversedBy: 'articles')]  
private Collection $Categories;
```

```
#[ORM\ManyToMany(targetEntity: Article::class, mappedBy: 'Categories')]  
private Collection $articles;
```

RELATION MANY TO MANY

Un constructeur a été rajouté pour initialiser l'attribut \$categories en un ArrayCollection()

un ArrayCollection() est une sorte de tableau php re-écrit en objet.

<https://www.doctrine-project.org/projects/doctrine-collections/en/1.6/index.html>

Il y a trois méthodes qui ont été rajoutées :

- getCategories() qui retourne la liste des catégories d'un article
- addCategory(Category \$category) qui rajoute une catégorie à l'article
- removeCategory(Category \$category) pour retirer une catégorie de l'article.

MISE À JOUR DATABASE

En mettant à jour, nous voyons que doctrine nous a créé une table intermédiaire pour la relation many to many ainsi que de nouveaux champs dans article et category pour l'utilisation de l'image.

C'est parfait!

```
PS D:\wamp64\www\sf4m2icours> php bin/console doctrine:schema:update --dump-sql

The following SQL statements will be executed:

    CREATE TABLE article_category (article_id INT NOT NULL, category_id INT NOT NULL, INDEX IDX_53A4EDAA7294869C (article_id
), INDEX IDX_53A4EDAA12469DE2 (category_id), PRIMARY KEY(article_id, category_id)) DEFAULT CHARACTER SET utf8mb4 COLLATE `utf
8mb4_unicode_ci` ENGINE = InnoDB;
    ALTER TABLE article_category ADD CONSTRAINT FK_53A4EDAA7294869C FOREIGN KEY (article_id) REFERENCES article (id) ON DELE
TE CASCADE;
    ALTER TABLE article_category ADD CONSTRAINT FK_53A4EDAA12469DE2 FOREIGN KEY (category_id) REFERENCES category (id) ON DE
LETE CASCADE;
    ALTER TABLE article ADD image_id INT DEFAULT NULL;
    ALTER TABLE article ADD CONSTRAINT FK_23A0E663DA5256D FOREIGN KEY (image_id) REFERENCES image (id);
    CREATE UNIQUE INDEX UNIQ_23A0E663DA5256D ON article (image_id);
    ALTER TABLE category ADD image_id INT DEFAULT NULL;
    ALTER TABLE category ADD CONSTRAINT FK_64C19C13DA5256D FOREIGN KEY (image_id) REFERENCES image (id);
    CREATE UNIQUE INDEX UNIQ_64C19C13DA5256D ON category (image_id);
PS D:\wamp64\www\sf4m2icours> php bin/console doctrine:schema:update --force

Updating database schema...

9 queries were executed
```

RELATION BI-DIRECTIONNELLE

Nous pouvons actuellement récupérer les catégories en passant par l'article et non l'inverse à moins de créer une méthode spécifique dans le repository (fichier de requête).

Nous allons faire de sorte que la relation entre Article et Category soit dans les deux sens, c'est-à-dire que l'on puisse récupérer les catégories d'un article (on l'a déjà) mais aussi les articles d'une catégorie, nous allons donc rendre la relation bidirectionnelle.

C'est l'occasion pour vous de découvrir les deux attributs mappedBy et inverseedBy. En fait, il y a toujours une entité qui est "propriétaire" de la relation.

L'entité propriétaire doit définir l'attribut "mappedBy" : il correspond à la propriété de l'objet "possédé" qui fait le lien entre les deux entités.

L'entité possédée doit définir l'attribut "inverseedBy" : il correspond à la propriété de l'objet "propriétaire" qui fait le lien entre les deux entités.

Et bien c'est déjà en place! Grace à la méthode CLI et votre choix :

```
Do you want to add a new property to Category so that you can access/update Article objects from it - e.g. $category->getArticles()? (yes/no) [yes]:  
>
```

```
A new property will also be added to the Category class so that you can access the related Article objects from it.  
New field name inside Category [articles]:  
>
```

DATA FIXTURE

data fixture permet de créer un jeu de données initial

```
D:\wamp64\www\sf4m2icours> composer require --dev orm-fixtures
```

Un dossier et fichier src/DataFixtures/AppFixtures.php sera créé.

```
namespace App\DataFixtures;

use App\Entity\Image;
use App\Entity\Category;
use Doctrine\Bundle\FixturesBundle\Fixture;
use Doctrine\Persistence\ObjectManager;

class AppFixtures extends Fixture
{
    public function load(ObjectManager $manager)
    {
        $image = new Image();
        $image->setChemin("https://via.placeholder.com/700x120/000000/FFFFFF/?text=Informations");
        $image->setAlt("Informations");
        $image->setIsPublished(true);

        $category = new Category();
        $category->setImage($image);
        $category->setLabel("Informations");
        $category->setContenu("Retrouvez l'ensemble des articles \"informations\"");
        $category->setIsPublished(true);

        $manager->persist($category);

        $manager->flush();
    }
}
```

DATA FIXTURE

```
$image = new Image();
$image->setChemin("https://via.placeholder.com/700x120/000000/FFFFFF/?text=Programmation");
$image->setAlt("Programmation");
$image->setIsPublished(true);

$category = new Category();
$category->setImage($image);
$category->setLabel("Programmation");
$category->setContenu("Retrouvez l'ensemble des articles \"Développement\"");
$category->setIsPublished(true);

$manager->persist($category);

$manager->flush();
}

}
```

Pour le lancer :

```
> Loading App\DataFixtures\AppFixtures
PS D:\wamp64\www\sf4m2icours> php bin/console doctrine:fixtures:load
```

DATA FIXTURE

Vous rencontrerez une erreur de violation de contrainte ou datecreation ne doit pas être null, pourtant vous avez ajouté une valeur par défaut dans l'ORM

Le tableau d'options et la valeur par défaut sont pour la création de schéma dans la base de données, pas pour fournir une valeur par défaut lors du vidage.
Nous devons donc modifier nos entités

image.php

```
/**  
 * @ORM\Column(type="datetime", options={"default": "CURRENT_TIMESTAMP"})  
 */  
private $dateCreation;  
  
public function __construct()  
{  
    $this->dateCreation = new \DateTime();  
}
```

category.php

```
/**  
 * @ORM\Column(type="datetime", options={"default": "CURRENT_TIMESTAMP"})  
 */  
private $dateCreation;  
  
public function __construct()  
{  
    $this->articles = new ArrayCollection();  
    $this->dateCreation = new \DateTime();  
}
```

Vous pouvez relancer la commande, cela s'ajoutera, vérifiez dans votre base de données!

Attention chaque chargement de fixture videra votre base de données

MANIPULER L'ORM

Le service Doctrine gère la persistance de nos objets. Il est accessible depuis le contrôleur (gestion connexion à la bdd, gestion des entitymanager) via l'injection de dépendances:

```
[ManagerRegistry $doctrine]
```

EntityManager c'est lui qui permet de dire à Doctrine « Persiste cet objet », c'est lui qui va exécuter les requêtes SQL (que l'on ne verra jamais):

```
$entityManager = $doctrine->getManager();
```

Les repositories sont des objets, qui utilisent un EntityManager en coulisses, mais qui sont bien plus faciles et pratiques à utiliser de notre point de vue. Je parle des repositories au pluriel car il en existe un par entité.

```
$imgRP = $entityManager->getRepository(Image::class);| You, il  
$categoryRP = $entityManager->getRepository(Category::class);
```

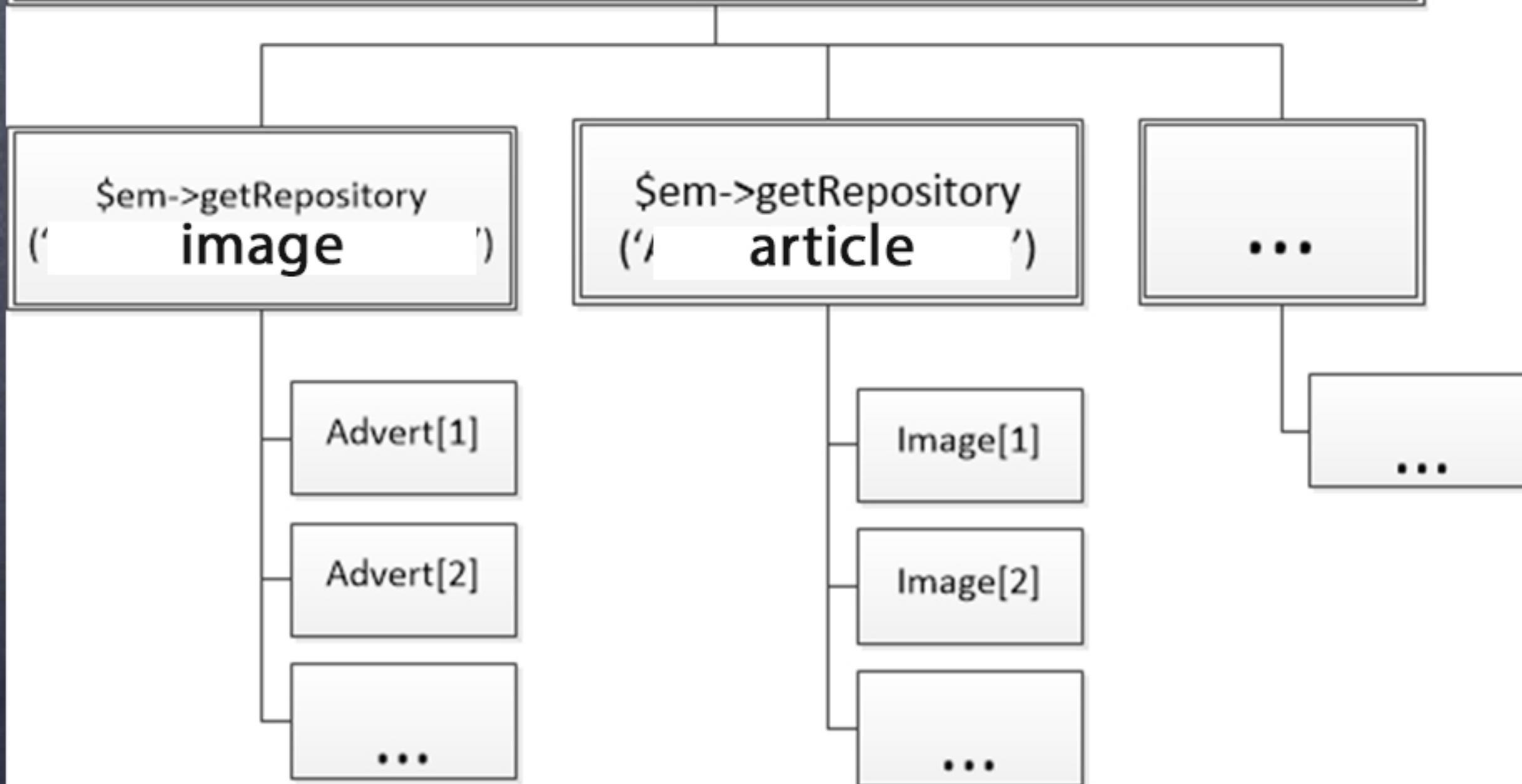
Ce sont donc ces repositories qui nous permettront de récupérer nos entités. Ainsi, pour charger deux entités différentes, il faut d'abord récupérer leur repository respectif.

MANIPULER L'ORM

schéma doctrine

ManagerRegistry \$doctrine
Par injection de dépendance

`$em = $doctrine->getManager()`



MANIPULER L'ORM

Je vais essayer de récupérer une image en base ainsi que les catégories, créer un nouvelle article et l'enregistrer en base. Dans blogController.php

```
#[Route(path: '/fixadd', name: 'fixadd')]
public function fixadd(ManagerRegistry $doctrine)
{
    $entityManager = $doctrine->getManager();

    $imgRP = $entityManager->getRepository(Image::class);
    $categoryRP = $entityManager->getRepository(Category::class);
    $img = $imgRP->findbyId(1);
    $category = $categoryRP->findAll();
    dump($category,$img);

    $article = new Article();
    $article->setTitle("Mon premier article");
    $article->setImage($img[0]);
    $article->addCategory($category[0]);
    $article->addCategory($category[1]);
    $article->setContent("Voici le contenu de mon article");
    $article->setLastUpdateDate(new \DateTime());
    $article->setIsPublished(false);

    $entityManager->persist($article);
    $entityManager->flush();
    dump($article);    You, il y a 2 minutes • Uncommitted char
    return new Response("<body>OK</body>");
}
```

Je récupère le repository d'image et category

Je réalise un findbyid (méthode interne) sur entity img

Je réalise un findByall (méthode interne) sur entity category

Je crée un nouvel article avec les setter associé

A noter category est de type arraycollection donc utilisation de la méthode «add»

Ensuite j'utilise persist pour initialiser la transaction et flush pour sauvegarder

MANIPULER L'ORM

Vous rencontrerez une erreur sur le champ «publicationDate» de l'article.
En effet comme pour l'image ou la catégorie il faut définir une valeur par défaut.

```
public function __construct()
{
    $this->Categories = new ArrayCollection();
    $this->publicationDate = new \DateTime();
}
```

Relancez la page.

Si vous avez oublié la création d'un champ n'hésitez pas à utiliser la commande

```
> php bin/console make:entity Article
PS D:\wamp64\www\sf6m2icours> php bin/console make:entity Article
Your entity already exists! So let's add some new fields!

New property name (press <return> to stop adding fields):
> lastUpdateDate

Field type (enter ? to see all types) [string]:
> datetime
```

MANIPULER L'ORM

Rendez-vous sur /fixadd

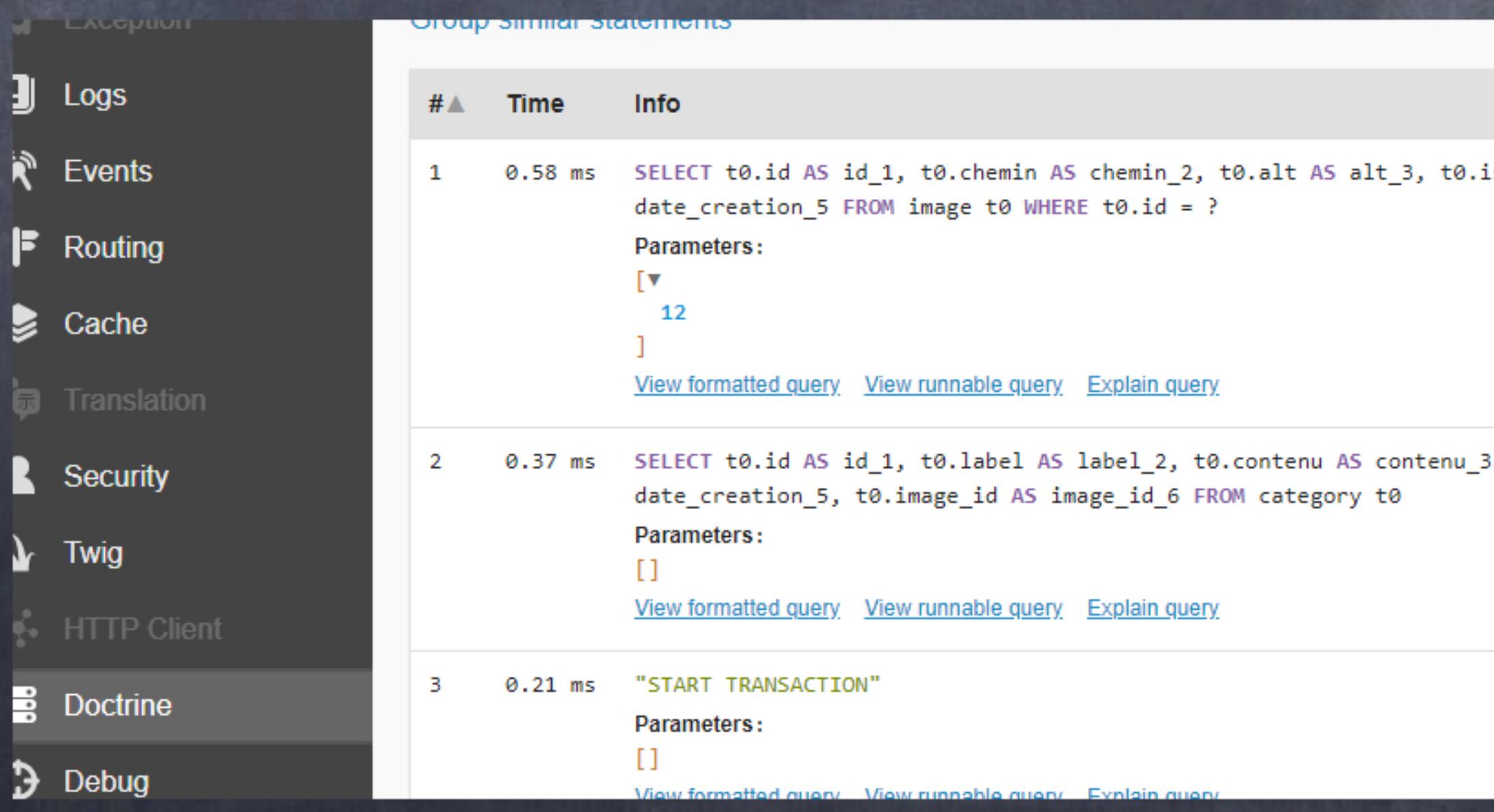


BlogController.php line 94
array:1 [▼
0 => App\Enti...\Image {#781 ▶}
]

BlogController.php line 94
array:2 [▼
0 => App\Enti...\Category {#792 ▶}
1 => App\Enti...\Category {#850 ▶}
]

sf4cours/_profiler/4c33fe?panel=dump 200 in 78.31 ms anon. 2 ms 7 in 2.46 ms + 2

Je peux visualiser mon dump de \$img et \$category et dans l'onglet doctrine du profiler je peux visualiser l'ensemble des requêtes



Logs
Events
Routing
Cache
Translation
Security
Twig
HTTP Client
Doctrine
Debug

Exception Group Similar Statements

#	Time	Info
1	0.58 ms	SELECT t0.id AS id_1, t0.chemin AS chemin_2, t0.alt AS alt_3, t0.is_date_creation_5 FROM image t0 WHERE t0.id = ? Parameters: [▼ 12] View formatted query View runnable query Explain query
2	0.37 ms	SELECT t0.id AS id_1, t0.label AS label_2, t0.contenu AS contenu_3, date_creation_5, t0.image_id AS image_id_6 FROM category t0 Parameters: [] View formatted query View runnable query Explain query
3	0.21 ms	"START TRANSACTION" Parameters: [] View formatted query View runnable query Explain query

MANIPULER L'ORM

Pourquoi deux méthodes `$em->persist()` et `$em->flush()`? Car cela permet entre autres de profiter des transactions. Imaginons que vous ayez plusieurs entités à persister en même temps. Par exemple, lorsque l'on crée un sujet sur un forum, il faut enregistrer l'entité-Sujet, mais aussi l'entité Message, les deux en même temps. Sans transaction, vous feriez d'abord la première requête, puis la deuxième. Logique au final. Mais imaginez que vous ayez enregistré votre Sujet, et que l'enregistrement de votre Message échoue : vous avez un sujet sans message ! Cela casse votre base de données, car la relation n'est plus respectée.

Avec une transaction, les deux entités sont enregistrées en même temps, ce qui fait que si la deuxième échoue, alors la première est annulée, et vous gardez une base de données propre.

La méthode `$em->persist()` permettant de donner la responsabilité de l'objet à doctrine, traite indifféremment les nouvelles entités de celles déjà en base de données. Vous pouvez donc lui passer une entité fraîchement créée ou une entité que vous auriez récupérée grâce au repository et que vous auriez modifiée (ou non, d'ailleurs). L'EntityManager s'occupe de tout.

Concrètement, cela veut dire que vous n'avez plus à vous soucier de faire des INSERT INTO dans le cas d'une création d'entité, et des UPDATE dans le cas d'entités déjà existantes.

MANIPULER L'ORM

Si vous visualisez votre base de données, vous voyez que doctrine n'a pas ajouté de nouvelles images ni de catégories mais associé simplement son id par contre si je lance :

```
[Route(path: '/fixadd', name: 'fixadd')]
public function fixadd(ManagerRegistry $doctrine)
{
    $entityManager = $doctrine->getManager();

    // $imgRP = $entityManager->getRepository(Image::class);
    $categoryRP = $entityManager->getRepository(Category::class);
    // $img = $imgRP->findId(1);
    $category = $categoryRP->findAll();
    dump($category, $img);

    $image = new Image();
    $image->setChemin("https://via.placeholder.com/700x120/000000/FFFFFF/?text=Programmation");
    $image->setAlt("Programmation");
    $image->setIsPublished(true);

    $article = new Article();
    $article->setTitle("Mon premier article");
    $article->setImage($image);
    $article->addCategory($category[0]);
    $article->addCategory($category[1]);
    $article->setContent("Voici le contenu de mon nouvel article"); You, il y a 1 seconde
    $article->setLastUpdateDate(new \DateTime());
    $article->setIsPublished(true);

    $entityManager->persist($article);
    $entityManager->flush();
    dump($article);

    return new Response("<body>OK</body>");
}
```

IL créera une nouvelle image en BDD et ajoutera son id à notre nouvelle article.

Veuillez noter que nous n'avons pas besoin de persister «image» car nous l'avons indiqué dans l'annotation de l'entité «cascade» : «persist»

MANIPULER L'ORM

Il est inutile de faire un `persist($entity)` lorsque \$entity a été récupérée grâce à Doctrine.

Un persist ne fait que donner la responsabilité d'un objet à Doctrine.

Dans le cas de la variable \$article de l'exemple précédent, Doctrine ne peut pas deviner qu'il doit s'occuper de cet objet si on ne le lui dit pas ! D'où le `persist()`.

LES AUTRES MÉTHODES DE ENTITY MANAGER

`detach($entity)` : annule le `persist()` effectué sur l'entité en argument. Au prochain `flush()`, aucun changement ne sera donc appliqué à l'entité

`clear($entity)` : annule tous les `persist()` effectués.

Si le nom d'une entité est précisé (son namespace complet ou son raccourci), seuls les `persist()` sur des entités de ce type seront annulés. Si `clear()` est appelé sans argument, cela revient à faire `undetach()` sur toutes les entités d'un coup.

`contains($entite)` : retourne true si l'entité donnée en argument est gérée par l'Entity Manager (s'il y a eu un `persist()` sur l'entité).

`refresh($entite)` : met à jour l'entité donnée en argument dans l'état où elle est en base de données. Cela écrase et donc annule tous les changements qu'il a pu y avoir sur l'entité concernée.

`remove($entite)` : supprime l'entité donnée en argument de la base de données. Effectif au prochain `flush()`.

LES REPOSITORIES

Un repository centralise tout ce qui touche à la récupération de vos entités.

Cela veut dire que vous ne devez pas faire la moindre requête SQL ailleurs que dans un repository.

On va y construire des méthodes pour récupérer une entité par son id, pour récupérer une liste d'entités suivant un critère spécifique, etc.

A chaque fois que vous devez récupérer des entités dans votre base de données, vous utiliserez le repository de l'entité correspondante.

Il existe un repository par entité. Cela permet de bien organiser son code. Bien sûr, cela n'empêche pas qu'un repository utilise plusieurs entités, dans le cas d'une jointure par exemple.

Pour info, il est possible de lancer des requêtes directement en commande via doctrine

```
PS D:\wamp64\www\sf6m2icours> php bin/console doctrine:query:sql "select * from image"
```

<code>id</code>	<code>chemin</code>	<code>alt</code>	<code>is_published</code>	<code>date_creation</code>
1	https://via.placeholder.com/700x120/000000/FFFFFF/?text=Informations	Informations	1	2022-08-12 09:13:48
2	https://via.placeholder.com/700x120/000000/FFFFFF/?text=Programmation	Programmation	1	2022-08-12 09:13:48
3	https://via.placeholder.com/700x120/000000/FFFFFF/?text=Programmation	Programmation	1	2022-08-12 17:46:23

```
PS D:\wamp64\www\sf6m2icours> php bin/console dbal:run-sql 'SELECT * FROM image'
```

<code>id</code>	<code>chemin</code>	<code>alt</code>	<code>is_published</code>	<code>date_creation</code>
1	https://via.placeholder.com/700x120/000000/FFFFFF/?text=Informations	Informations	1	2022-08-12 09:13:48

LES REPOSITORIES

Les méthodes de récupération de base :

Vos repositories héritent de la classe Doctrine\ORM\EntityRepository, qui propose déjà quelques méthodes très utiles pour récupérer des entités.

LES MÉTHODES NORMALES :

`find($id)` : La méthode `find($id)` récupère l'entité correspondante à l'id `$id`. Dans le cas de notre `ImageRepository`, elle retourne une instance d'`Image`.

`findAll()` : retourne toutes les entités contenue dans la base de données. Le format du retour est un tableau PHP normal (un array)

`findBy()` : Comme `findAll()`, elle permet de retourner une liste d'entités, sauf qu'elle est capable d'effectuer un filtre pour ne retourner que les entités correspondant à un ou plusieurs critère(s). Elle peut aussi trier les entités, et même n'en récupérer qu'un certain nombre (pour une pagination).

```
<?php  
$repository->findBy(  
    array $criteria,  
    array $orderBy = null,  
    $limit  = null,  
    $offset = null  
>;
```

```
$info = $entityManager->getRepository(Category::class)  
->findBy(["isPublished" => true],  
        ["id" => "desc"],  
        10,  
        0);  
dump($info);
```

LES REPOSITORIES

`findOneBy()` : La méthode `findOneBy(array $criteria, array $orderBy = null)` fonctionne sur le même principe que la méthode `findBy()`, sauf qu'elle ne retourne qu'une seule entité.

LES MÉTHODES MAGIQUES :

`findByX($valeur)` : Première méthode, en remplaçant « X » par le nom d'une propriété de votre entité. Dans notre cas, pour l'entité `Advert`, nous avons donc plusieurs méthodes :`findByTitle()`,`findByDate()`,`findByAuthor()`,`findByContent()`, etc.

Cette méthode fonctionne comme si vous utilisiez `findBy()` avec un seul critère, celui du nom de la méthode. [Retourne un tableau d'entity]

`findOneByX($valeur)` : Deuxième méthode, en remplaçant « X » par le nom d'une propriété de votre entité. Dans notre cas, pour l'entité `Advert`, nous avons donc plusieurs méthodes :`findOneByTitle()`,`findOneByDate()`,`findOneByAuthor()`,`findOneByContent()`, etc.

Cette méthode fonctionne comme `findOneBy()`, sauf que vous ne pouvez mettre qu'un seul critère, celui du nom de la méthode. [Retourne une entity]

LES REPOSITORIES

Mais que faire si vous avez besoin d'une requête plus complexe? Lorsque vous avez générée votre entité avec `make:entity`, la commande a également généré un `ArticleRepository` classe:

```
public function listingArticle($id):?array {
    $entityManager = $this->getEntityManager();

    $query = $entityManager->createQuery(
        'SELECT a
         FROM App\\Entity\\Article a
        WHERE a.id >= :id
        ORDER BY a.lastUpdateDate ASC'
    )->setParameter('id', $id);

    // returns an array of Product objects
    return $query->getResult();
}
```

```
$article = $entityManager->getRepository(Article::class)->listingArticle(1);
dump($article);
```

La chaîne passée à `createQuery()` pourrait ressembler à SQL, mais c'est Doctrine Query Language. Cela vous permet de taper des requêtes en utilisant un langage de requête connu, mais en référençant des objets PHP à la place

LES REPOSITORIES

Doctrine fournit également un query builder , un moyen orienté objet d'écrire des requêtes. Il est recommandé de l'utiliser lorsque les requêtes sont construites dynamiquement (c'est-à-dire en fonction des conditions PHP):
<https://www.doctrine-project.org/projects/doctrine-orm/en/current/reference/query-builder.html>

```
public function listingArticleQB($id, $publier = false):?array {
    $qb = $this->createQueryBuilder('a')
        ->where('a.id > :id')
        ->setParameter('id', $id)
        ->orderBy('a.lastUpdateDate', 'ASC');

    if(is_bool($publier)) {
        $qb->andWhere('a.isPublished = :published')
            ->setParameter('published', $publier);
    }

    $query = $qb->getQuery();

    return $query->execute();

    // Pour obtenir un seul résultat
    // $article = $query->setMaxResults(1)->getOneOrNullResult();
}
```

```
$article = $entityManager->getRepository(Article::class)->listingArticleQB(1);
dump($article);
```

LES REPOSITORIES

Il est possible également de réaliser du RAW SQL par contre vous n'obtiendrez pas des «objets»

A moins d'utiliser : NativeQuery

<https://www.doctrine-project.org/projects/doctrine-orm/en/current/reference/native-sql.html>

```
public function listingArticleRawSQL($id):mixed
{
    $conn = $this->getEntityManager()->getConnection();

    $sql = '
        SELECT *
        FROM Article
        WHERE id > :id
        ORDER BY last_update_date ASC
    ';
    $stmt = $conn->prepare($sql);
    $resultat = $stmt->executeQuery(['id' => $id]);

    // returns an array of arrays (i.e. a raw data set)
    return $resultat->fetchAllAssociative();
}

$article = $entityManager->getRepository(Article::class)->listingArticleRawSQL(1);
dump($article);
```

LES METHODES DU QUERY BUILDER

Le query builder est la référence dans la création de customs requêtes avec SF, voici ses méthodes de retour :

`getResult()` : Exécute la requête et retourne un tableau contenant les résultats sous forme d'objets. Vous récupérez ainsi une liste des objets, sur lesquels vous pouvez faire des opérations, des modifications, etc.

Même si la requête ne retourne qu'un seul résultat, cette méthode retourne un tableau.

`getArrayResult()` : Exécute la requête et retourne un tableau contenant les résultats sous forme de tableaux. Comme avec `getResult()`, vous récupérez un tableau même s'il n'y a qu'un seul résultat. Mais dans ce tableau, vous n'avez pas vos objets d'origine, vous avez des simples tableaux. Cette méthode est utilisée lorsque vous ne voulez que lire vos résultats

`getScalarResult()` : Exécute la requête et retourne un tableau contenant les résultats sous forme de valeurs. Comme avec `getResult()`, vous récupérez un tableau même s'il n'y a qu'un seul résultat.

Mais dans ce tableau, un résultat est une valeur, non un tableau de valeurs (`getArrayResult`) ou un objet de valeurs (`getResult`). Cette méthode est donc utilisée lorsque vous ne sélectionnez qu'une seule valeur dans la requête, par exemple :`SELECT COUNT(*) FROM ...` Ici, la valeur est la valeur du COUNT.

LES METHODES DU QUERY BUILDER

`getOneOrNullResult()` : Exécute la requête et retourne un seul résultat, ou null si pas de résultat. Cette méthode retourne donc une instance de l'entité (ou null) et non un tableau d'entités comme `getResult()`.

Cette méthode déclenche une exception `Doctrine\ORM\NonUniqueResultException` si la requête retourne plus d'un seul résultat.

`getSingleResult()` : Exécute la requête et retourne un seul résultat. Cette méthode est exactement la même que `getOneOrNullResult()`, sauf qu'elle déclenche une exception `Doctrine\ORM\NoResultException` si aucun résultat.

`getSingleScalarResult()` : Exécute la requête et retourne une seule valeur, et déclenche des exceptions si pas de résultat ou plus d'un résultat. (utile pour count)

`execute()` : Exécute la requête. Cette méthode est utilisée principalement pour exécuter des requêtes qui ne retournent pas de résultats (des UPDATE, INSERT INTO, etc.) :

LES JOINTURES

Lorsque vous utilisez la syntaxe `$entiteA->getEntiteB()`, Doctrine exécute une requête afin de charger les entités B qui sont liées à l'entité A.

L'objectif est donc de maîtriser quand charger juste l'entité A, et quand charger l'entité A avec ses entités B liées (lorsque nous sommes certains d'en avoir besoin).

Par exemple un `$repositoryA->find($id)` ne récupère qu'une seule entité A sans récupérer les entités liées.

Attention : on ne peut faire une jointure que si l'entité du FROM possède un attribut vers l'entité à joindre ! Cela veut dire que soit l'entité du FROM est l'entité propriétaire de la relation, soit la relation est bidirectionnelle.

LES JOINTURES

```
public function jointure($id,$publier=false):?array {  
    $qb = $this->createQueryBuilder('a')  
        ->leftJoin('a.image', "img")  
        ->addSelect('img')  
        //J'ai un C majuscule car mon attribut dans ma classe possède un C majuscule aussi  
        ->leftJoin('a.Categories', "cat")  
        ->addSelect('cat')  
        ->where('a.id >= :id')  
        ->setParameter('id', $id)  
        ->orderBy('a.lastUpdateDate', 'ASC');  
  
    if(is_bool($publier)) {  
        $qb->andWhere('a.isPublished = :published')  
            ->setParameter('published', $publier);  
    }  
  
    $query = $qb->getQuery();  
  
    return $query->getResult();  
}
```

On crée la jointure avec `leftJoin`. Le premier argument de la méthode est l'attribut de l'entité principale (celle qui est dans le `FROM` de la requête) sur lequel faire la jointure.

Dans l'exemple, l'entité Article possède un attribut `image`. Le deuxième argument de la méthode est l'alias de l'entité jointe (arbitraire).

Puis on sélectionne également l'entité jointe, via un `addSelect()`.

Et pourquoi n'a-t-on pas précisé la condition « ON » du JOIN ?

Doctrine connaît déjà tout sur notre association, grâce aux annotations ! Il est donc inutile de lui préciser le ON.

```
$article = $entityManager->getRepository(Article::class)->jointure(1);  
dump($article);
```