La rétro-conception de la base BMG

Comme le dit la documentation de Doctrine, faire du « reverse engineering » est un processus que l’on n’effectue qu’une seule fois lorsqu’on démarre un projet. Doctrine est capable de convertir environ 80% des informations de correspondance nécessaires en se basant sur les champs, les index et les contraintes de clés étrangères. **Il faut obligatoirement avoir des tables en InnoDB sous MySQL car le moteur MyISAM ne gère pas ces contraintes correctement**.

Il est parfois nécessaire de réaliser un **travail additionnel sur les entités générées** afin de satisfaire les spécificités du modèle.

 Le détail sur la rétro-conception des entités se trouve ici : <http://symfony.com/doc/current/doctrine.html>.

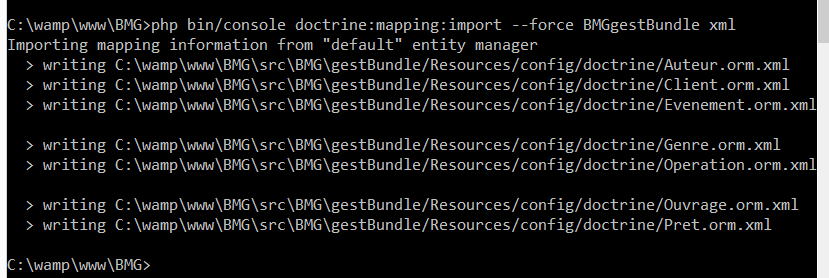
|  |  |
| --- | --- |
| Avant tout, assurez-vous que vos paramètres de connexion à la base de données bmg\_doctrine sont correctement définis dans le fichier *app/config/parameters.yml* |  |

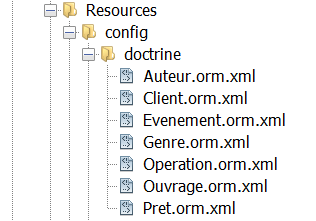
|  |  |
| --- | --- |
| La base de données bmg\_doctrine utilise dans la structure de ses tables des types de données non reconnus par Doctrine. Le champ "caution\_encaissee" de la table client est par exemple de type "bit".  Nous allons dans un premier temps, procéder à un petit réglage dans le fichier *config.yml* (type bit), afin que le type "bit" soit reconnu et converti en string. |  |

Nous pouvons à présent construire les classes entité depuis la base de données existante bmg\_doctrine. Demandons à Doctrine d'"**introspecter**" cette dernière et de générer les fichiers de méta-données correspondants en XML ( il existe d'autres possibilités que le XML).

php bin/console doctrine:mapping:import --force BMGgestBundle xml

Cette outil de ligne de commande demande à Doctrine d'introspecter la base de données et de générer les fichiers XML de méta-données dans le dossier *src/BMG/gestBundle/Resources/config/doctrine.*

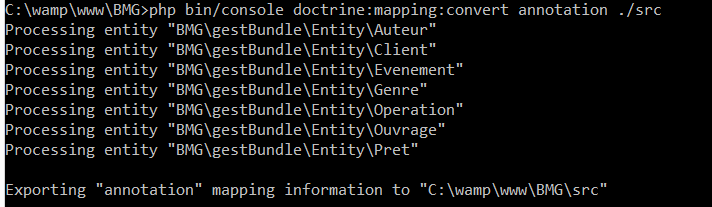


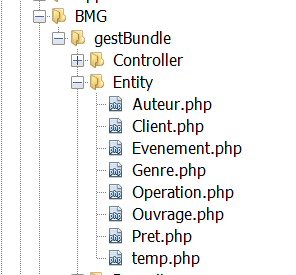
Voilà le résultat obtenu dans NetBeans : 

|  |  |
| --- | --- |
| Le fichier de méta-données *Genre.orm.xml* généré ressemble à ce qui suit : |  |

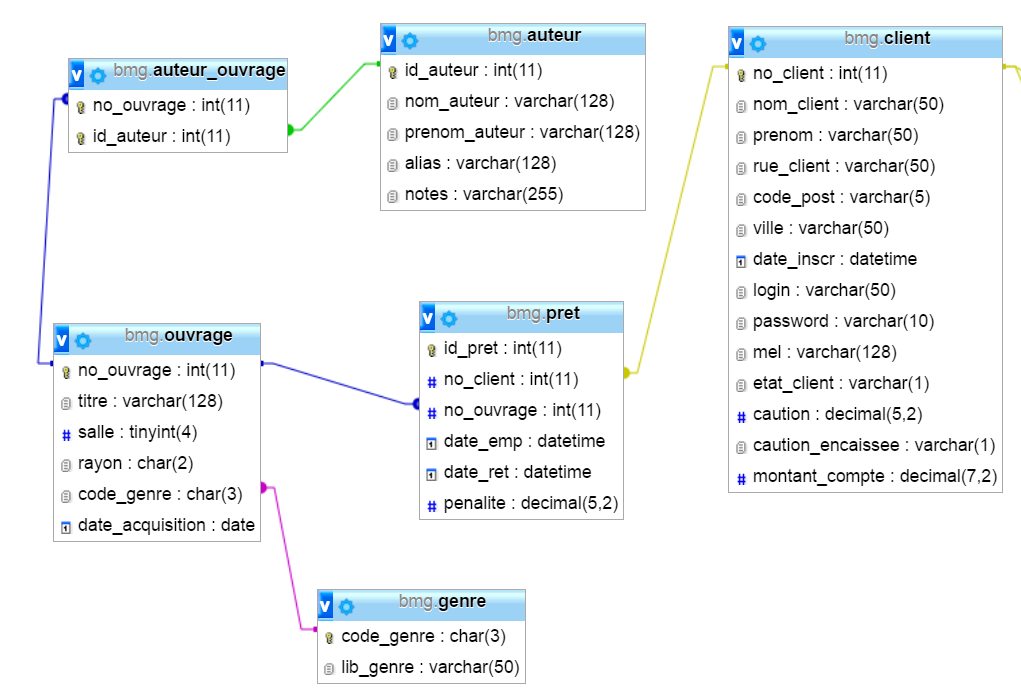
Une fois que les fichiers de méta-données sont générés, demandons à Doctrine d'importer le schéma et de construire les classes entité qui lui sont liées en exécutant la commande suivante :

php bin/console doctrine:mapping:convert annotation ./src



Les entités ont été créées : 

Nous allons à présent, observer chacune des classes créées, en les modifiant éventuellement pour les adapter plus finement à la base de données existante, dont le schéma est donné ci-dessous :



## **la classe Genre**

Doctrine a créé une classe Genre comportant deux attributs : le code (codeGenre) et le libellé (libGenre). codeGenre a bien été marqué comme identifiant (annotation @ORM\Id). Il faut néanmoins modifier la valeur de l'attribut strategy = "NONE" : l'identifiant n'est pas auto-incrémenté mais c'est à l'utilisateur de le saisir.

/\*\*

\* @var string

\*

\* @ORM\Column(name="code\_genre", type="string", length=3)

\* @ORM\Id

\* @ORM\GeneratedValue(strategy="NONE")

\*/

private $codeGenre;

## **Les classes Auteur et Ouvrage**

**La classe Auteur**

Selon le schéma relationnel, un auteur peut écrire plusieurs ouvrages et un ouvrage peut être écrit par plusieurs auteurs. En base de données, ceci se traduit par une table intermédiaire auteur\_ouvrage. Avec Doctrine, cette situation est représentée par une relation "*ManyToMany*".

Pour l'entité Auteur, Doctrine a créé un attribut de type collection Doctrine, qui est une collection des différents ouvrages créés par l'auteur. Cet attribut a été nommé $noOuvrage par Doctrine. Ce nom n'étant pas très représentatif, nous le renommerons **ouvrages**(avec 's' qui correspond à une liste d'ouvrages).

/\*\*

\* @var \Doctrine\Common\Collections\**Collection**

\*

\* @ORM\ManyToMany(targetEntity="BMG\gestBundle\Entity\**Ouvrage**", mappedBy="idAuteur")

\*/

private $ouvrages;

/\*\*

\* Constructor

\*/

public function \_\_construct()

{

$this->ouvrages = new \Doctrine\Common\Collections\ArrayCollection();

}

**Remarque:**

On remarquera qu'un constructeur a été créé pour initialiser la collection.

**La classe Ouvrage**

Parallèlement, dans l'entité Ouvrage, Doctrine a créé un attribut de type collection Doctrine, qui est une collection des différents auteurs de l'ouvrage. Cet attribut a été nommé $idAuteur par Doctrine. Ce nom n'étant pas très représentatif, nous le renommerons **auteurs**(avec 's' qui correspond à une liste d'auteurs).

/\*\*

\* @var \Doctrine\Common\Collections\**Collection**

\*

\* @ORM\ManyToMany(targetEntity="BMG\gestBundle\Entity\**Auteur**", inversedBy="noOuvrage")

\* @ORM\JoinTable(name="auteur\_ouvrage",

\* joinColumns={

\* @ORM\JoinColumn(name="no\_ouvrage", referencedColumnName="no\_ouvrage")

\* },

\* inverseJoinColumns={

\* @ORM\JoinColumn(name="id\_auteur", referencedColumnName="id\_auteur")

\* }

\* )

\*/

private $auteurs;

/\*\*

\* Constructor

\*/

public function \_\_construct()

{

$this->auteurs = new \Doctrine\Common\Collections\ArrayCollection();

}

**Remarques** :

Doctrine a créé ici une relation bidirectionnelle : l'entité Auteur contient la liste des ouvrages qu'a écrit l'auteur, l'entité ouvrage contient la liste des auteurs de l'ouvrage. Ceci permet de retrouver facilement la liste des ouvrages d'un auteur tout comme la liste des auteurs d'un ouvrage. Mais ce genre de relation doit s'accompagner d'un supplément de code lors de l'ajout d'un auteur ou d'un ouvrage, afin de garantir la cohérence de la relation. Nous verrons cela plus loin.

Ex : si l'ouvrage Germinal se trouve dans la collection d'ouvrages (ouvrages) de l'auteur Zola, il est évident que l'auteur Zola devra faire partie de la collection d'auteurs (auteurs) de l'ouvrage Germinal ☹ !!!

**Dernière chose** : une relation bidirectionnelle nécessite de spécifier les attributs permettant de lier les entités.

* Entité auteur : le lien se fait avec "ouvrages". La mention "inversed by" indique que l'entité auteur est **l'entité propriétaire.**

/\*\*

\* @var \Doctrine\Common\Collections\**Collection**

\*

\* @ORM\ManyToMany(targetEntity="BMG\gestBundle\Entity\**Auteur**", inversedBy="ouvrages")

\* @ORM\JoinTable(name="auteur\_ouvrage",

\* joinColumns={

\* @ORM\JoinColumn(name="no\_ouvrage", referencedColumnName="no\_ouvrage")

\* },

\* inverseJoinColumns={

\* @ORM\JoinColumn(name="id\_auteur", referencedColumnName="id\_auteur")

\* }

\* )

\*/

private $auteurs;

* Entité ouvrage : le lien se fait avec "auteurs". La mention "mappedBy" indique que l'entité ouvrage est **l'entité inverse.**

/\*\*

\* @var \Doctrine\Common\Collections\**Collection**

\*

\* @ORM\ManyToMany(targetEntity="BMG\gestBundle\Entity\**Ouvrage**", mappedBy="auteurs")

\*/

private $ouvrages;

/\*\*

\* Constructor

\*/

public function \_\_construct()

{

$this->ouvrages = new \Doctrine\Common\Collections\ArrayCollection();

}

Concernant l'entité Ouvrage, une relation ManyToOne relie cette entité avec l'entité Genre. Ouvrage se situe du côté Many et est donc propriétaire de la relation. Il en résulte qu'un attribut de la classe Genre (appelé "codeGenre" ) est présent dans le code. Renommons l'attribut "codeGenre" en "genre" afin d'être bien conscient qu'il ne s'agit pas juste du code du genre mais bien d'un **objet** Genre.

/\*\*

\* @var \BMG\gestBundle\Entity\Genre

\*

\* @ORM\ManyToOne(targetEntity="BMG\gestBundle\Entity\Genre")

\* @ORM\JoinColumns({

\* @ORM\JoinColumn(name="code\_genre", referencedColumnName="code\_genre")

\* })

\*/

private $genre;

## **La classe Pret**

Une relation ManyToOne existe entre l'entité Prêt et l'entité Ouvrage.

Une relation ManyToOne existe entre l'entité Prêt et l'entité Client.

Le côté Many se trouve, dans les deux cas du côté de l'entité Prêt. Cette entité contiendra donc un attribut Ouvrage et un attribut Client. Là encore, nous renommerons ces deux attributs afin de bien souligner le fait qu'il s'agit non pas respectivement d'un numéro d'ouvrage et d'un numéro de client mais bien d'un objet Ouvrage et d'un objet Client.

/\*\*

\* @var \BMG\gestBundle\Entity\Ouvrage

\*

\* @ORM\ManyToOne(targetEntity="BMG\gestBundle\Entity\Ouvrage")

\* @ORM\JoinColumns({

\* @ORM\JoinColumn(name="no\_ouvrage", referencedColumnName="no\_ouvrage")

\* })

\*/

private $ouvrage;

/\*\*

\* @var \BMG\gestBundle\Entity\Client

\*

\* @ORM\ManyToOne(targetEntity="BMG\gestBundle\Entity\Client")

\* @ORM\JoinColumns({

\* @ORM\JoinColumn(name="no\_client", referencedColumnName="no\_client")

\* })

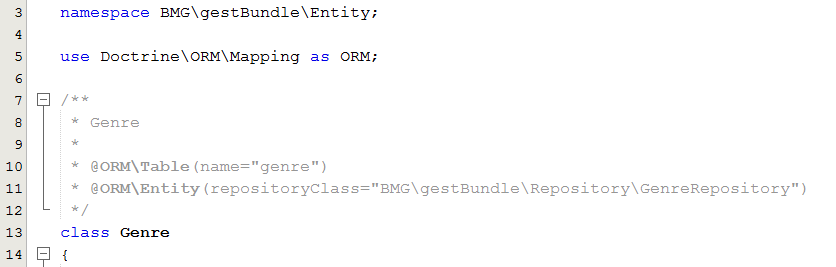
\*/

private $client;

## **Les repositories**

Nous créerons automatiquement les repositories grâce à la mention que nous allons ajouter dans chaque entité.

Voyons par exemple la mention à ajouter ligne 11, dans l'extrait du fichier genre.php. Le nom du Repository sera évidemment adapté à chaque classe.

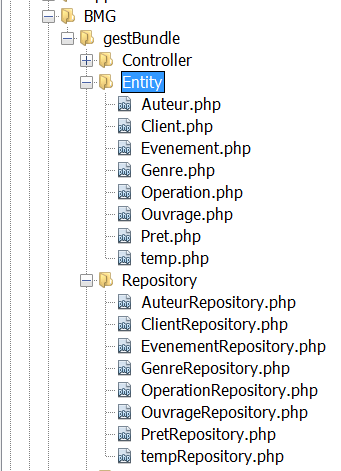


## **Générer automatiquement les accesseurs, les modificateurs et repository de chaque classe**

Nous allons à présent **générer les accesseurs, les modificateurs et repository** de chaque classe, en tapant en ligne de commande :

php bin/console doctrine:generate:entities BMGgestBundle

Vérifier dans chaque classe la présence des accesseurs et la création des différents repositories.



**Remarque** : on peut observer que pour tous les attributs de type collection (ex : ouvrages dans l'entité Auteur), les fonctions d'ajout, de suppression et d'obtention de la collection ont été ajoutées automatiquement.

/\*\*

\* Add auteur

\*

\* @param \BMG\gestBundle\Entity\Auteur $auteur

\*

\* @return Ouvrage

\*/

public function **addAuteur**(\BMG\gestBundle\Entity\Auteur **$auteur**)

{

*// le programmeur se souviendra qu'il devra utiliser cette méthode - addAuteur()- et // uniquement celle-là pour faire le lien entre auteur et ouvrage*

*// la méthode addOuvrage() de la classe Ouvrage sera proscrite car elle ne prend pas en*

*// compte la bidirectionnalité.*

$auteur->addOuvrage($this); *// à ajouter pour la birectionnalité*

$this->auteurs[] = $auteur;

return $this;

}

/\*\*

\* Remove auteur

\*

\* @param \BMG\gestBundle\Entity\Auteur $auteur

\*/

public function removeAuteur(\BMG\gestBundle\Entity\Auteur **$auteur**)

{

$this->auteurs->**removeElement**($auteur);

}

/\*\*

\* Get auteurs

\*

\* @return \Doctrine\Common\Collections\Collection

\*/

public function **getAuteurs**()

{

return $this->auteurs;

}

On peut à présent supprimer le dossier *src/BMG/gestBundle/Resources/config/doctrine* et son contenu : "If you want to use annotations, you must remove the XML (or YAML) files after running these two commands. This is necessary as [it is not possible to mix mapping configuration formats](http://symfony.com/doc/current/doctrine.html#doctrine-adding-mapping)".

Vos entités et repositories sont prêts à être utilisés.