M4D203D - POO Avancée

David Annebicque

2016

1 Sujet pour les TP

Proposer l'application qui permettra de jouer et de gérer le jeu (joueurs, parties, ...).

L'application devra permettre :

— Pouvoir gérer les cartes
— Pouvoir afficher les scores des joueurs
— Gérer les joueurs et les parties
— Sécuriser les parties et le back-office (prochaine séance)
— Obtenir un instantané d'une partie en cours

2 Les entités

2.1 Configuration de votre base de données

2.1.1 Le fichier parameters.yml

Ce fichier contient les informations sur votre connexion (et toutes les constantes de votre projet). Cous pourriez définir un fichier par environnement.

```
parameters:
    database_host:      localhost
    database_name:      test_project
    database_user:      root
    database_password:  password
```

2.1.2 Le fichier config.yml

C'est le fichier de configuration général de Symfony. Normalement la partie sur la base de données est déjà présente.

```
doctrine:
        dbal:
                driver:   pdo_mysql
                host:     '%database_host%'
                dbname:   '%database_name%'
                user:     '%database_user%'
                password: '%database_password%'
```

2.1.3 Création de la base de données

Suite à l'installation de Symfony et à la configuration des différents fichiers vous devez maintenant créer cette base de données sur votre serveur de base de données. Pour cela on utilise la console.

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

2.2 Création d'une entité liée à une table

Dans ce cas particulier notre entité sera une classe possédant des informations sur une table.

Rappel: Dans symfony une entité (ou modèle) est avant tout une classe PHP ! Cette entité n'est pas obligatoirement liée à une table. Cependant une table est Obligatoirement liée à une entité.

Pour créer une entité, il est conseillé d'utiliser la console pour avoir la structure minimale. La console n'est à utiliser qu'une seule fois par entité. Ensuite si vous souhaitez modifier une entité existante vous devez manipuler le fichier (ou le supprimer et repasser par la console).

```
php bin/console doctrine:generate:entity
```

Vous répondez à l'ensemble des questions.

— Le nom : Commence toujours par le nom de votre bundle puis ":" et le nom de votre entité (avec une majuscule)

— Le format : Vous choisirez annotation qui est le formalisme le plus simple dans un premier temps.

— les champs : Préciser le nom (minuscule), le type, la taille si besoin, si c'est un index (donc unique), s'il peut être null. Vous pourrez modifier dans le code ces éléments.
— validé en laissant le dernier champ vide

Vous ne devez pas ajouter la clé primaire. Dans Symfony elle est automatique et obligatoire. Par défaut elle se nomme "id".

A ce stade vous obtenez des classes qui contiennent des annotations pour créer et communiquer avec une table. Ce fichier est une classe. Vous pouvez donc ajouter des propriétés non liées à la table et des méthodes.

Vous constatez aussi que le fichier contient tous les getters et les setters nécessaires pour les propriétés définies.

```php
<?php
// src/AppBundle/Entity/Product.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
* @ORM\Entity
* @ORM\Table(name="product")
*/
class Product
{
        /**
        * @ORM\Column(type="integer")
        * @ORM\Id
        * @ORM\GeneratedValue(strategy="AUTO")
        */
        private $id;

        /**
        * @ORM\Column(type="string", length=100)
        */
        private $name;

        /**
        * @ORM\Column(type="decimal", scale=2)
        */
        private $price;

        /**
```

```
31          * @ORM\Column(type="text")
32          */
33          private $description;
34
35          //getters et setters
36          ...
37  }
```

Si vous ajoutez des champs manuellement (en ajouter une propriété private et son annotation), il vous faudra aussi écrire les getters et setters associés. Pour cela, vous pouvez vous aider de la console en exécutant la commande :

```
1  php bin/console doctrine:generate:entities AppBundle:Product
```

Vous pouvez préciser l'entité concernée. Si vous ne précisez que le bundle, l'ensemble des entités seront parcourus et les getters et setters manquant seront ajoutés.

Vous pouvez maintenant mettre à jour votre base de données en exécutant la commande :

```
1  php bin/console doctrine:schema:update --force
```

Vous devez exécuter cette commande à chaque modification de vos entités afin que les mise à jour soient faite sur votre base de données.

3 Les relations entre les entités

Avec les étapes précédentes il est possible de créer autant de table qu'on le souhaite. Cependant ces tables n'ont aucun lien entre-elles. Et l'intérêt d'une base de données se trouve donc limité. Grâce à Doctrine (ou n'importe quel ORM), nous allons pouvoir définir les liens qui existent entre les entités, ce qui se traduira par des clés étrangères dans les tables.

Il existe 4 relations dans Doctrine:

— OneToMany - Une instance de l'entité courante a plusieurs instances de l'entité cible (1-n)

— ManyToOne - Plusieurs instances de l'entité courante font références à une instance de l'entité cible (n-1)

— OneToOne - Une instance de l'entité courante fait référence à une instance de l'entité cible (1-1)

— ManyToMany (collections) - Plusieurs instances de l'entité courante font références à plusieurs instances de l'entité cible (n-n)

3.1 Many-To-One, Unidirectional

C'est l'association la plus courante dans un projet. L'exemple ci dessous indique que l'utilisateur possède une adresse et qu'une adresse peut être possédée par plusieurs utilisateurs.

```php
<?php
...
/** @Entity */
class User
{
        // ...

        /**
        * Many Users have One Address.
        * @ORM\ManyToOne(targetEntity="Address")
        * @ORM\JoinColumn(name="address_id",
 ↪  referencedColumnName="id")
        */
        private $address;
}

/** @Entity */
class Address
{
        // ...
}
```

L'annotation JoinColumn est optionnelle et permet de nommer la relation. Par défaut elle correspond au nom de l'entité suivi de la clé primaire.

3.2 One-To-One, Unidirectional

Cette association permet de lier un produit à un transporteur (shipping). Cette relation est unidirectionnelle c'est à dire qu'on peut connaitre le transporteur à partir du produit, mais on ne peut pas connaitre le produit lié au transporteur.

On arrive ici sur une notion importante de doctrine. La notion de direction. Les liaisons sont au minimum unidirectionnelles c'est à dire qu'elles fonctionnent de manière classique et selon la conception imaginée dans le MCD/MLD. Cependant il est possible de mettre en place des relations bi-directionnelles qui vont permettre

de récupérer les informations en partant de la source. Nous verrons cela par la suite.

```php
<?php
/** @Entity */
class Product
{
        // ...

        /**
        * One Product has One Shipping.
        * @ORM\OneToOne(targetEntity="Shipping")
        * @ORM\JoinColumn(name="shipping_id",
 ↪  referencedColumnName="id")
        */
        private $shipping;

        // ...
}

/** @Entity */
class Shipping
{
        // ...
}
```

3.3 One-To-One, Bidirectional

La relation bi-directionnelle implique de définir la liaison sur les deux entités. Dans ce cas il est obligatoire de préciser qui porte la relation (mappedBy) et qui est l'inverse de la relation (inversedBy). **La valeur à mettre dans cette propriété est le nom de la propriété PHP (la variable) dans la classe target (sans le $)**.

```php
<?php
/** @Entity */
class Customer
{
        // ...

        /**
        * One Customer has One Cart.
```

```
9           * @ORM\OneToOne(targetEntity="Cart",
    ↪  mappedBy="customer")
10          */
11          private $cart;
12
13          // ...
14  }
15
16  /** @Entity */
17  class Cart
18  {
19          // ...
20
21          /**
22          * One Cart has One Customer.
23          * @ORM\OneToOne(targetEntity="Customer",
    ↪  inversedBy="cart")
24          * @ORM\JoinColumn(name="customer_id",
    ↪  referencedColumnName="id")
25          */
26          private $customer;
27
28          // ...
29  }
```

3.4 One-To-Many, Bidirectional

La relation Many-to-one possède aussi une relation inverse. Celle-ci se nomme One-To-Many. En fait, on n'utilise jamais le One-To-Many seul. Il est toujours dans une relation bi-directionnelle. La logique est la même que pour une relation bi-directionnelle one-to-one.

```
1   <?php
2   use Doctrine\Common\Collections\ArrayCollection;
3
4   /** @Entity */
5   class Product
6   {
7           // ...
8           /**
9           * One Product has Many Features.
```

```
10          * @OneToMany(targetEntity="Feature",
    ↪  mappedBy="product")
11          */
12          private $features;
13          // ...
14
15          public function __construct() {
16                  $this->features = new ArrayCollection();
17          }
18  }
19
20  /** @Entity */
21  class Feature
22  {
23          // ...
24          /**
25          * Many Features have One Product.
26          * @ManyToOne(targetEntity="Product",
    ↪  inversedBy="features")
27          * @JoinColumn(name="product_id",
    ↪  referencedColumnName="id")
28          */
29          private $product;
30          // ...
31  }
```

3.5 Many-To-Many, Bidirectional

L'instruction Many-to-many est toujours (en tout cas pour ce qui nous concerne) bi-directionnelle. Il faut donc définir ou choisir qui porte la relation et qui l'inverse. Noter que dans ce cas, plusieurs objets sont liées à plusieurs autres objets. Cela implique l'apparition des tableaux dans les constructeurs pour contenir la "collection" d'objets associés à notre objet courant. Les getters et les setters sont également légèrement différents et introduisent des notions de add, remove et get.

```
1   <?php
2   /** @Entity */
3   class User
4   {
5           // ...
```

```
6
7           /**
8           * Many Users have Many Groups.
9           * @ManyToMany(targetEntity="Group",
    ↪  inversedBy="users")
10          * @JoinTable(name="users_groups")
11          */
12          private $groups;
13
14          public function __construct() {
15                  $this->groups = new
                    ↪  \Doctrine\Common\Collections\ArrayCollection();
16          }
17
18          // ...
19  }
20
21  /** @Entity */
22  class Group
23  {
24          // ...
25          /**
26          * Many Groups have Many Users.
27          * @ManyToMany(targetEntity="User",
    ↪  mappedBy="groups")
28          */
29          private $users;
30
31          public function __construct() {
32                  $this->users = new
                    ↪  \Doctrine\Common\Collections\ArrayCollection();
33          }
34
35          // ...
36  }
```