Développez votre site web avec le framework Symfony2 (ancienne version)

30 heures Facile Licence

LA COUCHE MÉTIER : LES ENTITÉS

L'objectif d'un ORM (pour *Object-Relation Mapper*, soit en français « lien objet-relation ») est simple : se charger de l'enregistrement de vos données en vous faisant oublier que vous avez une base de données. Comment ? En s'occupant de tout ! Nous n'allons plus écrire de requêtes, ni créer de tables via phpMyAdmin. Dans notre code PHP, nous allons faire appel à Doctrine2, l'ORM par défaut de Symfony2, pour faire tout cela.

Notions d'ORM : fini les requêtes, utilisons des objets

Je vous propose de commencer par un exemple pour bien comprendre. Supposons que vous disposiez d'une variable `$utilisateur`, un objet `User` qui représente l'un de vos utilisateurs qui vient de s'inscrire sur votre site. Pour sauvegarder cet objet, vous êtes habitués à créer votre propre fonction qui effectue une requête SQL du type `INSERT INTO` dans la bonne table, etc. Bref, vous devez gérer tout ce qui touche à l'enregistrement en base de données. En utilisant un ORM, vous n'aurez plus qu'à utiliser quelques fonctions de cet ORM, par exemple : `$orm->save($utilisateur)`. Et ce dernier s'occupera de tout ! Vous avez enregistré votre utilisateur en une seule ligne. Bien sûr, ce n'est qu'un exemple, nous verrons les détails pratiques dans la suite de ce chapitre, mais retenez bien l'idée.

Mais l'effort que vous devrez faire pour bien utiliser un ORM, c'est d'oublier votre côté « administrateur de base de données ». Oubliez les requêtes SQL, pensez objet !

Vos données sont des objets

Dans ORM, il y a la lettre O comme Objet. En effet, pour que tout le monde se comprenne, toutes vos données doivent être sous forme d'objets. Concrètement, qu'est-ce que cela implique dans notre code ? Pour reprendre l'exemple de notre utilisateur, quand vous étiez petits, vous utilisiez sûrement un tableau, puis vous accédiez à vos attributs via `$utilisateur['email']` par exemple. Soit, c'était très courageux de votre part. Mais nous allons aller plus loin, maintenant.

Utiliser des objets n'est pas une grande révolution en soi. Faire `$utilisateur->getEmail()` au lieu de `$utilisateur['email']`, c'est joli, mais limité. Ce qui est une révolution, c'est de coupler cette représentation objet avec l'ORM. Qu'est-ce que vous pensez d'un `$utilisateur->getCommentaires()` ? Ha ha ! vous ne pouviez pas faire cela avec votre tableau ! Ici, la méthode `$utilisateur->getCommentaires()` déclencherait la bonne requête, récupérerait tous les commentaires postés par votre utilisateur, et vous retournerait une sorte de tableau d'objets de type `Commentaire` que vous pourriez afficher sur la page de profil de votre utilisateur, par exemple. Cela commence à devenir intéressant, n'est-ce pas ?

Au niveau du vocabulaire, un objet dont vous confiez l'enregistrement à l'ORM s'appelle une **entité** (*entity* en anglais). On dit également **persister** une entité, plutôt qu'enregistrer une entité. Vous savez, l'informatique et le jargon…

Créer une première entité avec Doctrine2

Une entité, c'est juste un objet

Derrière ce titre se cache la vérité. Une entité, ce que l'ORM va manipuler et enregistrer dans la base de données, ce n'est vraiment rien d'autre qu'un simple objet. Voici ce à quoi pourrait ressembler l'objet `Advert` de notre plateforme d'annonces :

php

```php
<?php
// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

class Advert
{
  protected $id;

  protected $content;

  // Et bien sûr les getters/setters :

  public function setId($id)
  {
    $this->id = $id;
  }
  public function getId()
  {
    return $this->id;
  }

  public function setContent($content)
  {
    $this->content = $content;
  }
  public function getContent()
  {
    return $this->content;
  }
}
```

Inutile de créer ce fichier pour l'instant, nous allons le générer plus bas, patience.

Comme vous pouvez le voir, c'est très simple. Un objet, des propriétés, et bien sûr, les *getters/setters* correspondants. On pourrait en réalité utiliser notre objet dès maintenant !

php

```php
<?php
// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use OC\PlatformBundle\Entity\Advert;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class AdvertController extends Controller
{
  public function viewAction()
  {
    $advert = new Advert;
    $advert->setContent("Recherche développeur Symfony2.");

    return $this->render('OCPlatformBundle:Advert:view.html.twig', array(
      'advert' => $advert
    ));
  }
}
```

Et voilà notre objet Advert est opérationel. Bien sûr, l'exemple est un peu limité car statique, mais l'idée est là et vous voyez comment l'on peut se servir d'une entité. Retenez donc : une entité n'est rien d'autre qu'un objet.

Normalement, vous devez vous poser une question : comment l'ORM va-t-il faire pour enregistrer cet objet dans la base de données s'il ne connaît rien de nos propriétés `id` et `content` ? Comment peut-il deviner que notre propriété id doit être stockée dans une colonne de type `INT` dans la table ? La réponse est aussi simple que logique : il ne devine rien, on va le lui dire !

Une entité, c'est juste un objet… mais avec des commentaires !

Quoi ? Des commentaires ?

OK, je dois avouer que ce n'est pas intuitif si vous ne vous en êtes jamais servi, mais oui, on va ajouter des commentaires dans notre code et Symfony2 va se servir directement de ces commentaires pour ajouter des fonctionnalités à notre application. Ce type de commentaires se nomme l'**annotation**. Les annotations doivent respecter une syntaxe particulière, regardez par vous-mêmes :

php

```php
<?php
// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

// On définit le namespace des annotations utilisées par Doctrine2
// En effet, il existe d'autres annotations, on le verra par la suite,
```

```php
// qui utiliseront un autre namespace
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class Advert
{
  /**
   * @ORM\Column(name="id", type="integer")
   * @ORM\Id
   * @ORM\GeneratedValue(strategy="AUTO")
   */
  protected $id;

  /**
   * @ORM\Column(name="date", type="date")
   */
  protected $date;

  /**
   * @ORM\Column(name="title", type="string", length=255)
   */
  protected $title;

  /**
   * @ORM\Column(name="author", type="string", length=255)
   */
  protected $author;

  /**
   * @ORM\Column(name="content", type="text")
   */
  protected $content;

  // Les getters
  // Les setters
}
```

Ne recopiez toujours pas toutes ces annotations à la main, on utilise le générateur en console au paragraphe juste en dessous.

Attention par contre pour les prochaines annotations que vous serez amenés à écrire à la main : elles doivent être dans des commentaires de type « `/**` », avec précisément deux étoiles. Si vous essayez de les mettre dans un commentaire de type « `/*` » ou encore « `//` », elles seront simplement ignorées.

Grâce à ces annotations, Doctrine2 dispose de toutes les informations nécessaires pour utiliser notre objet, créer la table correspondante, l'enregistrer, définir un identifiant (id) en auto-incrément, nommer les colonnes, etc. Ces informations se nomment les *metadata* de notre entité. Je ne vais pas épiloguer sur les annotations, elles sont suffisamment claires pour être comprises par tous.  Ce qu'on vient de faire, à savoir rajouter les *metadata* à notre objet `Advert`, s'appelle *mapper* l'objet `Advert`. C'est-à-dire faire le lien entre notre objet de base et la représentation

physique qu'utilise Doctrine2.

Sachez quand même que, bien que l'on utilisera les annotations tout au long de ce cours, il existe d'autres moyens de définir les *metadata* d'une entité : en YAML, en XML et en PHP. Si cela vous intéresse, vous trouverez plus d'informations sur la définition des *metadata* via les autres moyens dans le chapitre Doctrine2 de la documentation de Symfony2.

Générer une entité : le générateur à la rescousse !

En tant que bon développeur, on est fainéant à souhait, et ça, Symfony2 l'a bien compris ! On va donc se refaire une petite session en console afin de générer notre première entité. Entrez la commande suivante et suivez le guide :

```
C:\wamp\www\Symfony>php app/console generate:doctrine:entity

  Welcome to the Doctrine2 entity generator


This command helps you generate Doctrine2 entities.

First, you need to give the entity name you want to generate.
You must use the shortcut notation like AcmeBlogBundle:Post.

The Entity shortcut name:_
```

Grâce à ce que le générateur vous dit, vous l'avez compris, il faut entrer le nom de l'entité sous le format `NomBundle:NomEntité`. Dans notre cas, on entre donc `OCPlatformBundle:Advert`.

```
The Entity shortcut name: OCPlatformBundle:Advert

Determine the format to use for the mapping information.

Configuration format (yml, xml, php, or annotation) [annotation]:_
```

Comme je vous l'ai dit, nous allons utiliser les annotations, qui sont d'ailleurs le format par défaut. Appuyez juste sur la touche **Entrée**.

```
Configuration format (yml, xml, php, or annotation) [annotation]:

Instead of starting with a blank entity, you can add some fields now.
Note that the primary key will be added automatically (named id).

Available types: array, simple_array, json_array, object,
boolean, integer, smallint, bigint, string, text, datetime, datetimetz,
date, time, decimal, float, blob, guid.

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

On commence à saisir le nom de nos champs. Lisez bien ce qui est inscrit avant : Doctrine2 va ajouter

automatiquement l'id, de ce fait, pas besoin de le redéfinir ici. On entre donc notre date : `date`.

```
New field name (press <return> to stop adding fields): date
Field type [string]:_
```

C'est maintenant que l'on va dire à Doctrine à quel type correspond notre propriété `date`. La liste des types possibles vous est donné par Symfony juste au dessus. Nous voulons une date avec les informations de temps, tapez donc `datetime`.

5. Répétez les points 3 et 4 pour les propriétés `title`, `author` et `content`. `title` et `author` sont de type `string` de 255 caractères (pourquoi pas). `Content` est par contre de type `text`.

```
New field name (press <return> to stop adding fields): date
Field type [string]: datetime

New field name (press <return> to stop adding fields): title
Field type [string]: string
Field length [255]: 255

New field name (press <return> to stop adding fields): author
Field type [string]: string
Field length [255]: 255

New field name (press <return> to stop adding fields): content
Field type [string]: text

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

Lorsque vous avez fini, appuyez sur la touche **Entrée**.

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

Do you want to generate an empty repository class [no]?_
```

Oui, on va créer le *repository* associé, c'est très pratique, nous en reparlerons largement. Entrez donc `yes`.

8. Confirmez la génération, et voilà !

```
Do you want to generate an empty repository class [no]? yes


  Summary before generation


You are going to generate a "OCPlatformBundle:Advert" Doctrine2 entity
using the "annotation" format.

Do you confirm generation [yes]?
```

```
  Entity generation

Generating the entity code: OK

  You can now start using the generated code!


C:\wamp\www\Symfony>_
```

Allez tout de suite voir le résultat dans le fichier `Entity/Advert.php`. Symfony2 a tout généré, même les *getters* et les *setters* ! Vous êtes l'heureux propriétaire d'une simple classe… avec plein d'annotations !

On a utilisé le générateur de code pour nous faciliter la vie. Mais sachez que vous pouvez tout à fait vous en passer ! Comme vous pouvez le voir, le code généré n'est pas franchement compliqué, et vous pouvez bien entendu l'écrire à la main si vous préférez.

Affiner notre entité avec de la logique métier

L'exemple de notre entité Advert est un peu simple, mais rappelez-vous que la couche modèle dans une application est la couche métier. C'est-à-dire qu'en plus de gérer vos données un modèle contient également la logique de l'application. Voyez par vous-mêmes avec les exemples ci-dessous.

Attributs calculés

Prenons l'exemple d'une entité `Commande`, qui représenterait un ensemble de produits à acheter sur un site d'e-commerce. Cette entité aurait les attributs suivants :

- `ListeProduits` qui contient un tableau des produits de la commande ;
- `AdresseLivraison` qui contient l'adresse où expédier la commande ;
- `Date` qui contient la date de la prise de la commande ;
- Etc.

Ces trois attributs devront bien entendu être *mappés* (c'est-à-dire définis comme des colonnes pour l'ORM via des annotations) pour être enregistrés en base de données par Doctrine2. Mais il existe d'autres caractéristiques pour une commande, qui nécessitent un peu de calcul : le prix total, un éventuel coupon de réduction, etc. Ces caractéristiques n'ont pas à être persistées en base de données, car elles peuvent être déduites des informations que l'on a déjà. Par exemple, pour avoir le prix total, il suffit de faire une boucle sur `ListeProduits` et d'additionner le prix de chaque produit :

php

```php
<?php
// Exemple :
class Commande
{
```

```
  public function getPrixTotal()
  {
    $prix = 0;
    foreach($this->getListeProduits() as $produit)
    {
      $prix += $produit->getPrix();
    }
    return $prix;
  }
}
```

N'hésitez donc pas à créer des méthodes `getQuelquechose()` qui contiennent de la logique métier. L'avantage de mettre la logique dans l'entité même est que vous êtes sûrs de réutiliser cette même logique partout dans votre application. Il est bien plus propre et pratique de faire `<?php $commande->getPrixTotal()` que d'éparpiller à droite et à gauche différentes manières de calculer ce prix total. Bien sûr, ces méthodes n'ont pas d'équivalent `setQuelquechose()`, cela n'a pas de sens !

Attributs par défaut

Vous avez aussi parfois besoin de définir une certaine valeur à vos entités lors de leur création. Or nos entités sont de simples objets PHP, et la création d'un objet PHP fait appel… au constructeur. Pour notre entité `Advert`, on pourrait définir le constructeur suivant :

php

```
<?php
// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Advert
 *
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\AdvertRepository")
 */
class Advert
{
  // ...

  public function __construct()
  {
    // Par défaut, la date de l'annonce est la date d'aujourd'hui
    $this->date = new \Datetime();
  }

  // ...
}
```

Conclusion

N'oubliez pas : une entité est un objet PHP qui correspond à un besoin dans votre application.

N'essayez donc pas de raisonner en termes de tables, base de données, etc. Vous travaillez maintenant avec des objets

PHP, qui contiennent une part de logique métier, et qui peuvent se manipuler facilement. C'est vraiment important que vous fassiez l'effort dès maintenant de prendre l'habitude de manipuler des objets, et non des tables.

Tout sur le mapping !

Vous avez rapidement vu comment *mapper* vos objets avec les annotations. Mais ces annotations permettent d'inscrire pas mal d'autres informations. Il faut juste en connaître la syntaxe, c'est l'objectif de cette section.

Tout ce qui va être décrit ici se trouve bien entendu dans la documentation officielle sur le mapping, que vous pouvez garder à portée de main.

L'annotation `Entity`

L'annotation `Entity` s'applique sur une classe, il faut donc la placer avant la définition de la classe en PHP. Elle définit un objet comme étant une entité, et donc persisté par Doctrine. Cette annotation s'écrit comme suit :

```
@ORM\Entity
```

Il existe un seul paramètre facultatif pour cette annotation, `repositoryClass`. Il permet de préciser le namespace complet du repository qui gère cette entité. Nous donnerons le même nom à nos repositories qu'à nos entités, en les suffixant simplement de « Repository ». Pour notre entité `Advert`, cela donne :

text
```
1 @ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\AdvertRepository")
```

Un *repository* sert à récupérer vos entités depuis la base de données, on en reparle dans un chapitre dédié plus loin dans le cours.

L'annotation `Table`

L'annotation `Table` s'applique sur une classe également. C'est une annotation facultative, une entité se définit juste par son annotation `Entity`. Cependant, l'annotation `Table` permet de personnaliser le nom de la table qui sera créée dans la base de données. Par exemple, on pourrait préfixer notre table `advert` par « oc » :

text
```
1 @ORM\Table(name="oc_advert")
```

Elle se positionne juste avant la définition de la classe.

Par défaut, si vous ne précisez pas cette annotation, le nom de la table créée par Doctrine2 est le même que celui de l'entité. Dans notre cas, cela aurait été « Advert », avec la majuscule donc, attention car la convention de nommage des tables d'une base de données est de ne pas employer de majuscule. Pensez aussi que si vous êtes sous Windows cela n'a pas d'importance, mais quand vous déploierez votre site sur un serveur sous Linux, bonjour les erreurs de casse !

L'annotation `Column`

L'annotation `Column` s'applique sur un attribut de classe, elle se positionne donc juste avant la définition PHP de

l'attribut concerné. Cette annotation permet de définir les caractéristiques de la colonne concernée. Elle s'écrit comme suit :

```
@ORM\Column
```

L'annotation `Column` comprend quelques paramètres, dont le plus important est le type de la colonne.

Les types de colonnes

Les types de colonnes que vous pouvez définir en annotation sont des types Doctrine, *et uniquement Doctrine*. Ne les confondez pas avec leurs homologues SQL ou PHP, ce sont des types à Doctrine seul. Ils font la transition des types SQL aux types PHP.

Voici dans le tableau suivant la liste exhaustive des types Doctrine2 disponibles.

Type Doctrine	Type SQL	Type PHP	Utilisation
string	VARCHAR	string	Toutes les chaînes de caractères jusqu'à 255 caractères.
integer	INT	integer	Tous les nombres jusqu'à 2 147 483 647.
smallint	SMALLINT	integer	Tous les nombres jusqu'à 32 767.
bigint	BIGINT	string	Tous les nombres jusqu'à 9 223 372 036 854 775 807. Attention, PHP reçoit une chaîne de caractères, car il ne supporte pas un si grand nombre (suivant que vous êtes en 32 ou en 64 bits).
boolean	BOOLEAN	boolean	Les valeurs booléennes `true` et `false`.
decimal	DECIMAL	double	Les nombres à virgule.
date ou datetime	DATETIME	objet DateTime	Toutes les dates et heures.
time	TIME	objet DateTime-	Toutes les heures.
text	CLOB	string	Les chaînes de caractères de plus de 255 caractères.
object	CLOB	Type de l'objet stocké	Stocke un objet PHP en utilisant `serialize` / `unserialize`.
array	CLOB	array	Stocke un tableau PHP en utilisant `serialize` / `unserialize`.

float	FLOAT	double	Tous les nombres à virgule. Attention, fonctionne uniquement sur les serveurs dont la locale utilise un point comme séparateur.

Les types Doctrine sont sensibles à la casse. Ainsi, le type « String » n'existe pas, il s'agit du type « string ». Facile à retenir : tout est en minuscule !

Le type de colonne se définit en tant que paramètre de l'annotation `Column`, comme suit :

```
@ORM\Column(type="string")
```

Les paramètres de l'annotation `Column`

Il existe 7 paramètres, tous facultatifs, que l'on peut passer à l'annotation `Column` afin de personnaliser le comportement. Voici la liste exhaustive dans le tableau suivant.

Paramètre	Valeur par défaut	Utilisation
type	string	Définit le type de colonne comme nous venons de le voir.
name	Nom de l'attribut	Définit le nom de la colonne dans la table. Par défaut, le nom de la colonne est le nom de l'attribut de l'objet, ce qui convient parfaitement. Mais vous pouvez changer le nom de la colonne, par exemple si vous préférez « isExpired » en attribut, mais « is_expired » dans la table.
length	255	Définit la longueur de la colonne. Applicable uniquement sur un type de colonne `string`.
unique	`false`	Définit la colonne comme unique. Par exemple sur une colonne e-mail pour vos membres.
nullable	`false`	Permet à la colonne de contenir des `NULL`.
precision	0	Définit la précision d'un nombre à virgule, c'est-à-dire le nombre de chiffres en tout. Applicable uniquement sur un type de colonne `decimal`.
scale	0	Définit le *scale* d'un nombre à virgule, c'est-à-dire le nombre de chiffres après la virgule. Applicable uniquement sur un type de colonne `decimal`.

Pour définir plusieurs options en même temps, il faut simplement les séparer avec une virgule. Par exemple, pour une colonne « e-mail » en `string` 255 et unique, il faudra faire :

```
@ORM\Column(type="string", length=255, unique=true)
```

Pour conclure

Vous savez maintenant tout ce qu'il faut savoir sur la couche Modèle sous Symfony2 en utilisant les entités de l'ORM Doctrine2.

Je vous redonne l'adresse de la documentation Doctrine2, que vous serez amenés à utiliser maintes fois dans vos développements : http://docs.doctrine-project.org/proje [...] st/index.html. J'insiste : enregistrez-la dans vos favoris, car Doctrine est une bibliothèque très large, et bien que je vous donne un maximum d'informations dans cette partie du cours, je ne pourrai pas tout couvrir.

Attention, Doctrine étant une bibliothèque totalement indépendante de Symfony2, sa documentation fait référence à ce type d'annotation : `/** @Entity **/`.
Il faut impérativement l'adapter à votre projet Symfony2, en préfixant toutes les annotations par « ORM\ » comme nous l'avons vu dans ce chapitre : `/** @ORM\Entity **/`.
Car dans nos entités, c'est le namespace ORM que nous chargeons. Ainsi, l'annotation `@Entity` n'existe pas pour nous, c'est `@ORM` qui existe (et tous ses enfants : `@ORM\Entity`, `@ORM\Table`, etc.).

Dans le prochain chapitre, nous apprendrons à manipuler les entités que nous savons maintenant construire.

En résumé

- Le rôle d'un ORM est de se charger de la persistance de vos données : vous manipulez des objets, et lui s'occupe de les enregistrer en base de données.
- L'ORM par défaut livré avec Symfony2 est Doctrine2.
- L'utilisation d'un ORM implique un changement de raisonnement : on utilise des objets, et on raisonne en POO. C'est au développeur de s'adapter à Doctrine2, et non l'inverse !
- Une entité est, du point de vue PHP, un simple objet. Du point de vue de Doctrine, c'est un objet complété avec des informations de *mapping* qui lui permettent d'enregistrer correctement l'objet en base de données.
- Une entité est, du point de vue de votre code, un objet PHP qui correspond à un besoin, et indépendant du reste de votre application.

Activité : Créez votre CoreBundle pour les gouverner tous

Manipuler ses entités avec Doctrine2

L'auteur

Alexandre Bacco

Découvrez aussi ce cours en...