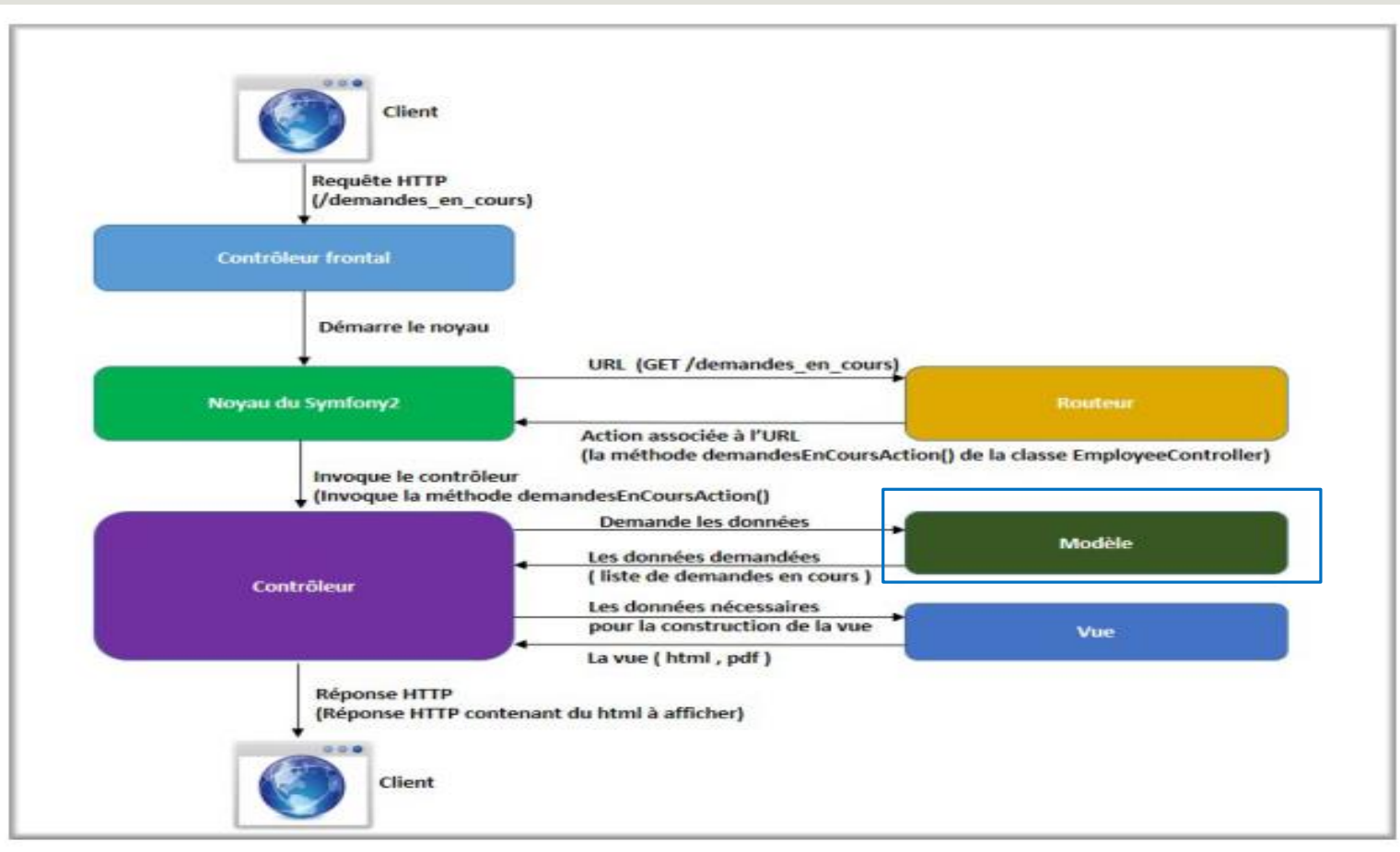


Symfony

L'ORM DOCTRINE

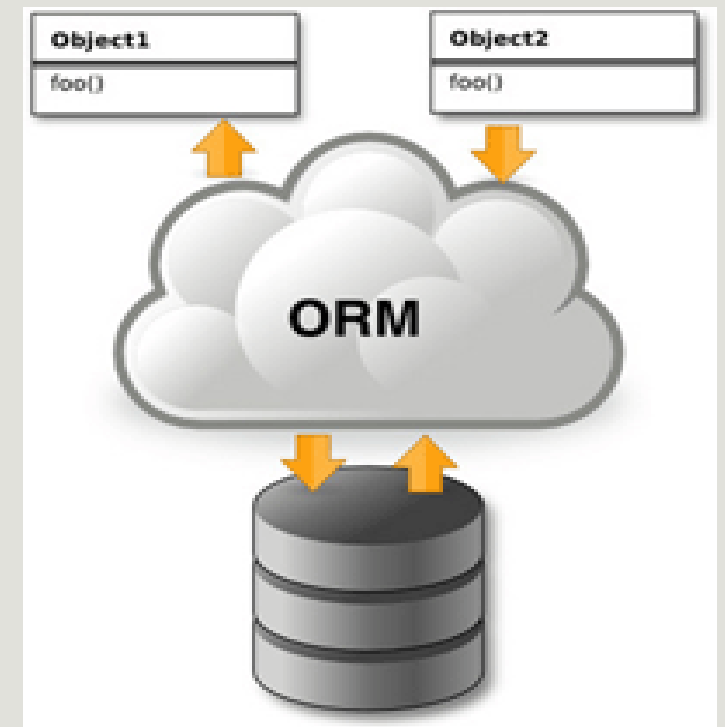
AYMEN SELLAOUTI

Introduction (1)

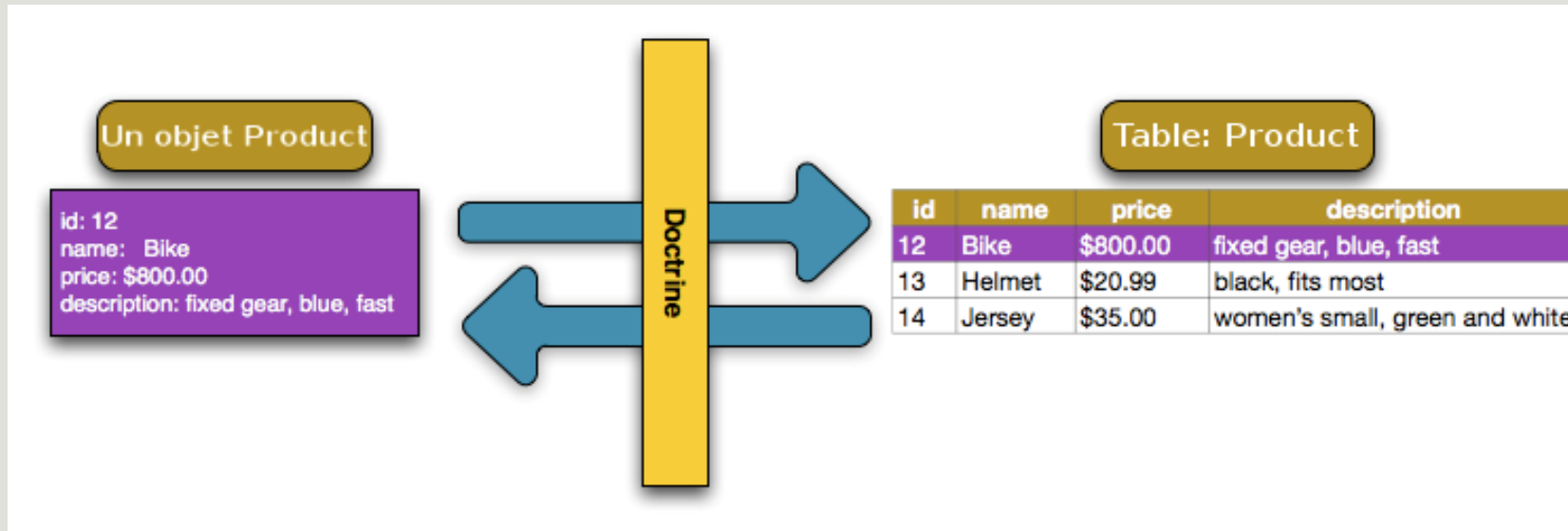


Introduction (2)

- ORM : Object Relation Mapper
- Couche d'abstraction
- Gérer la persistance des données
- Mapper les tables de la base de données relationnelle avec des objets
- Crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé.
- Propose des méthodes prédéfinies



Introduction (3)



- Doctrine : ORM le plus utilisé avec Symfony2
- Associe des classes PHP avec les tables de votre BD (mapping)

Les entités (1)

- Objet PHP
- Les entités représente les objets PHP équivalentes à une table de la base de données.
- Une entité est généralement composée par les attributs de la tables ainsi que leurs getters et setters
- Manipulable par l'ORM

Les entités (2)

Exemple

```
<?php

namespace Rt4\AsBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Etudiant
 *
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="Rt4\AsBundle\Entity\EtudiantRepository")
 */
class Etudiant
{
    /**
     * @var integer
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;
```

```
/**
 * @var integer
 *
 * @ORM\Column(name="numEtudiant", type="integer")
 */
private $numEtudiant;

/**
 * @var integer
 *
 * @ORM\Column(name="cin", type="integer")
 */
private $cin;

/**
 * @var string
 *
 * @ORM\Column(name="nom", type="string", length=255)
 */
private $nom;

/**
 * @var string
 *
 * @ORM\Column(name="prenom", type="string", length=255)
 */
```

```
private $prenom;

/**
 * @var \DateTime
 *
 * @ORM\Column(name="dateNaissance", type="date")
 */
private $dateNaissance;

/**
 * Get id
 *
 * @return integer
 */
public function getId()
{
    return $this->id;
}

/**
 * Set numEtudiant
 *
 * @param integer $numEtudiant
 * @return Etudiant
 */
```

```
/**
 * Set numEtudiant
 *
 * @param integer $numEtudiant
 * @return Etudiant
 */
public function setNumEtudiant($numEtudiant)
{
    $this->numEtudiant = $numEtudiant;

    return $this;
}

/**
 * Get numEtudiant
 *
 * @return integer
 */
public function getNumEtudiant()
{
    return $this->numEtudiant;
}
```

Configuration des entités

Configuration Externe : YAML, XML, PHP

Configuration Interne : annotations

Choix de la configuration ?

Deux Visions :

Pro-Externe

Séparation complète des fonctionnalités spécialement lorsque l'entité est conséquente

Pro-Interne

Plus facile et agréable de chercher dans un seul fichier l'ensemble des informations, plus de visibilité

Mapping : Annotation des entités (1)

Rôle : Faire le lien entre les entités et les tables de la base de données

Lien à travers les **métadonnées**

Remarque : Un seul format par bundle (impossibilité de mélanger)

Syntaxe :

/**

***** les différentes annotations

***/**

Remarque : Afin d'utiliser les annotations il faut ajouter :

```
use Doctrine\ORM\Mapping as ORM;
```


Mapping : Annotation des entités (2)

Entity

Permet de définir un `objet` comme une `entité`

Applicable sur une `classe`

Placée avant la définition de la classe en PHP

Syntaxe :

```
@ORM\Entity
```

Paramètres :

`repositoryClass` (facultatif). Permet de préciser le namespace complet du repository qui gère cette entité.

Exemple :

```
@ORM\Entity(repositoryClass="Rt4\AsBundle\Entity\animalRepository")
```

Mapping : Annotation des entités (3)

Table

Permet de spécifier le nom de la table dans la base de données à associer à l'entité

Applicable sur une classe et placée avant la définition de la classe en PHP

Facultative sans cette annotation le nom de la table sera automatiquement le nom de l'entité

Généralement utilisable pour ajouter des préfixes ou pour forcer la première lettre de la table en minuscule

Syntaxe : `@ORM\Table()`

Exemple :

```
/**
 *
 * @ORM\Table('animal')
 * @ORM\Entity(repositoryClass="Rt4\AsBundle\Entity\animalRepository")
 */
```

Mapping : Annotation des entités (4)

Column

Permet de définir les caractéristiques de la colonne concernée

Applicable sur un attribut de classe juste avant la définition PHP de l'attribut concerné.

Syntaxe : `@ORM\Column()`

Exemple :

```
/**  
 *  
 * @ORM\Column(param1="valParam1",param2="valParam2")  
 */
```

Mapping : Annotation des entités (5)

Les paramètres de Column

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
length	255	Définit la longueur de la colonne (pour les strings).
unique	false	Définit la colonne comme unique (Exemple : email).
nullable	false	Permet à la colonne de contenir des NULL.
precision	0	Définit la précision d'un nombre à virgule(decimal)
scale	0	le nombre de chiffres après la virgule (decimal)

Mapping : Annotation des entités (6)

Les types de Column

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.
boolean	BOOLEAN	boolean	Les valeurs booléennes true et false.
decimal	DECIMAL	double	Les nombres à virgule.

Mapping : Annotation des entités (7)

Les types de Column

Type Doctrine	Type SQL	Type PHP	Utilisation
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.

Mapping : Annotation des entités (8)

Conventions de Nommage

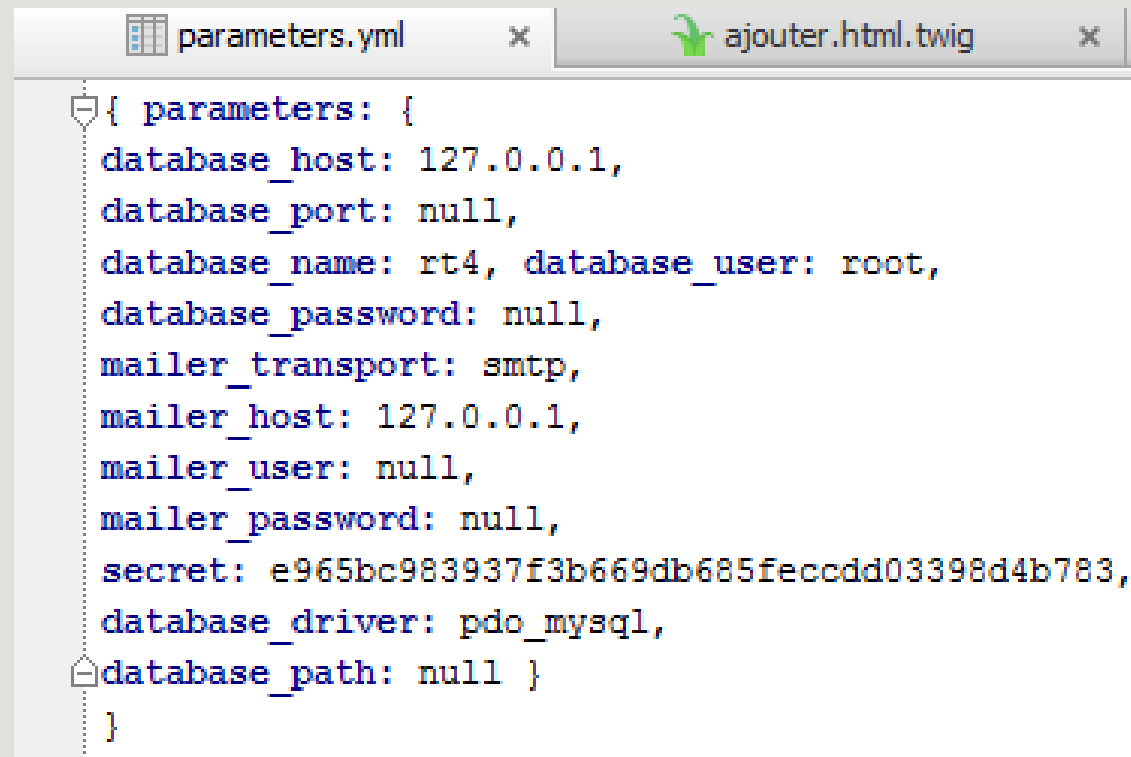
Même s'il reste facultatif, le champs « name » doit être modifié afin de respecter les conventions de nommage qui diffèrent entre ceux de la base de données et ceux de la programmation OO.

- Les noms de classes sont écrites en « Pascal Case » TheEntity.
- Les attributs de classes sont écrites en « camel Case » oneAttribute
- Les noms des tables et des colonnes en SQL sont écrites en minuscules, les mots sont séparés par « _ » one_table, one_column.

Gestion de la base de données (1)

Configuration de l'application

Afin de configurer la base de données de l'application il faut renseigner les champs dans le fichier [parameters.yml](#)



```
{ parameters: {  
  database_host: 127.0.0.1,  
  database_port: null,  
  database_name: rt4, database_user: root,  
  database_password: null,  
  mailer_transport: smtp,  
  mailer_host: 127.0.0.1,  
  mailer_user: null,  
  mailer_password: null,  
  secret: e965bc983937f3b669db685feccdd03398d4b783,  
  database_driver: pdo_mysql,  
  database_path: null }  
}
```


Gestion de la base de données (2)

Création de base de données

Afin de créer la base de données du projet 2 méthodes sont utilisées :

- Manuelle en utilisant le SGBD (le nom de la BD doit être le même que celui mentionné dans le fichier parameters.yml)
- En utilisant la ligne de command avec la command suivante :
 - `php app/console doctrine:database:create`
 - Une base de données avec les propriétés mentionnées dans parameters.yml sera automatiquement générée

Gestion de la base de données (3)

Génération des entités

Deux méthodes pour générer les entités :

- Méthode manuelle (non recommandée)
 - Créer la classe
 - ajouter le mapping
 - ajouter les getters et les setters (manuellement ou en utilisant la commande suivante :
`php app/console doctrine:generate:entities` (elle crée les getters et les setters de toutes les entités)
- Méthode en utilisant les commandes
 - Il suffit de lancer la commande suivante :
`php app/console doctrine:generate:entity`
 - Ajouter les attributs ainsi que les paramètres qui vont avec
 - Une fois terminé, Doctrine génère l'entité avec toutes les métadonnées de mapping

Gestion de la base de données (4)

Création des tables de la base de données

Afin de créer les tables de la base de données doctrine se base sur les entités placées dans les différents dossiers Entity des différents bundles de l'application.

- Deux commandes permettent de créer les tables de la BD :
 - `php app/console doctrine:schema:create`
 - `php app/console doctrine:schema:update --force //` utilisable pour la création et pour la mise à jour d'une table

Astuce :

- `php app/console doctrine:schema:update --dump-sql //` Affiche les requêtes SQL à exécuter pour la BD

Cette astuce permet de vérifier la requête à exécuter avant la mise à jour de la table.

Le service Doctrine

Rôle : permet la gestion des données dans la BD : **persistance** des données et **consultation** des données.

Méthode :

- `$this->get('doctrine');`
- `$this->getDoctrine();` //helper (raccourcie de la classe Controller)

Le service Doctrine offre deux services pour la gestion d'une base de données :

- Le Repository qui se charge des requêtes Select
- L'EntityManager qui se charge de persister la base de données donc de gérer les requêtes INSERT, UPDATE et DELETE.

Le Repository

Les repositories (dépôts)

Des classes PHP dont le rôle est de permettre à l'utilisateur de récupérer des entités d'une classe donnée.

Syntaxe :

Pour accéder au repository de la classe Maclasse on utilise l'EntityManager

```
$repo = $EntityManager->getRepository(« Bundle:MAClasse »);
```

Quelque méthodes offertes par le repository :

```
$repository->findAll(); // récupère tous les entités (enregistrements) relatifs à l'entité associé au repository
```

```
$repository->find($id); // requête sur la clé primaire
```

```
$repository->findBy(); // retourne un ensemble d'entités avec un filtrage sur plusieurs critères (nbre donné)
```

```
$repository->findOneBy(); // même principe que findBy mais une seule entité
```

```
$repository->findByNomPropriété(); $repository->findOneByNomPropriété();
```

Le Repository

findAll

findAll()

Rôle : retourne l'ensemble des entités qui correspondent à l'entité associée au repository. Le format du retour est un Array

Exemple :

//On récupère le repository de l'entity manager correspondant à l'entité Etudiant

```
$repository = $this
```

```
->getDoctrine()
```

```
->getRepository('OCPlatformBundle:Advert') ;
```

//On récupère la liste des étudiants

```
$listAdverts = $repository->findAll();
```

Généralement le tableau obtenu est passé à la vue (TWIG) et est affiché en utilisant un foreach

Le Repository find

`find($id)`

Rôle : retourne l'entité qui correspond à la clé primaire passé en argument. Généralement cette clé est l'id.

Exemple :

```
//On récupère le repository de l'entity manager correspondnat à l'entité Etudiant
```

```
$repo = $this
```

```
->getDoctrine()
```

```
->getRepository('Rt4AsBundle:Etudiant');
```

```
//on lance la requête sur l'étudiant d'id 2
```

```
$etudiant = $repository->find(2);
```

Le Repository findBy

findBy()

Rôle : retourne l'ensemble des entités qui correspondent à l'entité associé au repository comme findAll sauf qu'elle permet d'effectuer un filtrage sur un ensemble de critère passés dans un Array. Elle offre la possibilité de trier les entités sélectionnées et facilite la pagination en offrant un nombre de valeur de retour.

Syntaxe:

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

Exemple :

```
$repository = $this->getDoctrine()->getRepository('Rt4AsBundle:Etudiant');  
$listeEtudiants = $repository->findBy(array('section' => 'RT4','nom' => 'Mohamed'),  
    array('date' => 'desc'),10, 0);
```


Le Repository findOneBy

findOneBy()

Rôle : Même principe ue FindBy mais en retournant une seule entité ce ui élimine automatiquement les paramètres d'ordre de limite et d'offset

Exemple :

```
$repository = $this->getDoctrine()->getRepository('Rt4AsBundle:Etudiant');  
$Etud = $repository->findOneBy(array('section' => 'RT4','nom' => 'Mohamed'));
```

Le Repository

findByPropriété

findByPropriété()

Rôle : En remplaçant le mot **Propriété** par le **nom d'une des propriété de l'entité**, la fonction va faire le même rôle que **findBy** mais avec un seul critère qui est le nom de la propriété et sans les options.

Exemple :

```
$repository = $this->getDoctrine()->getRepository('Rt4AsBundle:Etudiant');  
$listeEtudiants = $repository->findByNom('Aymen');
```

Le Repository

findOneByPropriété

findOneByPropriété()

Rôle : En remplaçant le mot **Propriété** par le **nom d'une des propriété de l'entité**, la fonction va faire le même rôle que **findOneBy** mais avec un seul critère.

Exemple :

```
$repository = $this->getDoctrine()->getRepository('Rt4AsBundle:Etudiant');  
$listeEtudiants = $repository->findOneByNom('Aymen');
```

Le Repository

Création de requêtes

Les requêtes de doctrine sont écrites en utilisant le langage de doctrine le Doctrine Query Language **DQL** ou en utilisant un Objet créateur de requêtes le **CreateQueryBuilder**

createQuery : Méthode de l'Entity Manager

```
$query = $em->createQuery(
    'SELECT e
     FROM Rt4AsBundle:Etudiant e
     WHERE e.age > :age
     ORDER BY e.age ASC')
    ->setParameter('age', '22');
$etudiants = $query->getResult();
```

➤ **CreateQueryBuilder** : Méthode du repository

```
$repository=$this->getEntityManager()->getRepository('Rt4AsBundle:Etudiant');
$query = $repository->createQueryBuilder('e')
    ->where('e.age > :age')
    ->setParameter('age', '22')
    ->orderBy('e.age', 'asc')
    ->getQuery();
$etudiants = $query->getResult();
```

Le Repository

CreateQuery et DQL

Le DQL peut être défini comme une adaptation du SQL adapté à l'orienté objet et donc à DOCTRINE

La requête est défini sous forme d'une chaîne de caractère

Afin de créer une requête DQL il faut utiliser la méthode `createQuery()` de l'EntityManager

La méthode `setParameter('label','valeur')` permet de définir un paramètre de la requête

Pour définir plusieurs paramètres ou bien utiliser `setParameter` plusieurs fois ou bien la méthode `setParameters(array('label1','valeur1', 'label2','valeur2'),.. 'labelN','valeurN'))`

Une fois la requête créée la méthode `getResult()` permet de récupérer un tableau de résultat

Le langage DQL est explicité dans le lien suivant :

<http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/dql-doctrine-query-language.html>

Le Repository QueryBuilder

Constructeur de requête DOCTRINE Alternative au DQL

Accessible via le **Repository**

Le résultat fourni par la méthode **getQuery** du **QueryBuilder** permet de **générer la requête en DQL**

De même que le **createQuery**, une fois la requête créée la méthode **getResult()** permet de récupérer un tableau de résultat.

```
$repository=$this->getEntityManager()->getRepository('Rt4AsBundle:Etudiant');  
$query = $repository->createQueryBuilder('e')  
    ->where('e.age > :age')  
    ->setParameter('age','22')  
    ->orderBy('e.age','asc')  
    ->getQuery();  
$etudiants = $query->getResult();
```

<http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/query-builder.html>

Le Repository QueryBuilder

- Afin de récupérer le QueryBuilder dans notre Repository, on utilise la méthode **createQueryBuilder**.
- Cette méthode récupère en paramètre l'alias de l'entité cible et offre la requête « select from » de l'entité en question.
- Généralement l'alias est la première lettre en minuscule du nom de l'entité
- Si aucun paramètre n'est passé à createQueryBuilder alors on aura une requête vide et il faudra tout construire.

```
$qb=$this->_em->createQueryBuilder()  
    ->select('t')  
    ->from($this->_entityName,'alias');
```

```
$qb=$this->createQueryBuilder('t')
```

Le Repository QueryBuilder : Méthodes

- **from('entityName','entityAlias')**
 - `from($this->_entityName,'t')`
- **where('condition')** permet d'ajouter le where dans la requête
 - `where('t.destination= :dest')`
- **setParameter('nomParam',param)** permet d'ajouter la définition d'un des paramètres définis dans le where
 - `setParameter('dest',$dest)`
- **andWhere('condition')** permet d'ajouter d'autres conditions
 - `andWhere('t.statut = :status')`
- **orderBy('nomChamp','ordre')** permet d'ajouter un orderBy et prend en paramètre le champ à ordonner et l'ordre DESC ou ASC.
 - `orderBy('t.dateTransfert','DESC')`
- **setParameters(array(1=>'param1',2=>'param2'))**

Le Repository

Externalisation des requêtes dans un dépôt personnalisé pour chaque entité

But :

- Isoler la couche modèle
- Réutilisabilité

Faisabilité :

Ajouter le dépôt dans le mapping de l'entité (@ORM\Entity(repositoryClass=« NotreRepository »))

Exemple :

```
/**
 * Etudiant
 *
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="Rt4\AsBundle\Entity\EtudiantRepository")
 */
class Etudiant
{
```

Le service EntityManager

Rôle : L'interface ORM proposée par doctrine offrant des méthodes prédéfini pour persister dans la base ou pour mettre à jour ou supprimer une entité.

Méthode :

- `$EntityManager = $this->get('doctrineorm.entity_manager')`
- `$this->getDoctrine()->getManager(); //helper (raccourcie de la classe Controller)`

Remarque il y a aussi la méthode `getEntityManager()` mais elle est devenu obsolète.

Le service EntityManager

Insertion des données

Enregistrement des données

Etant un ORM, Doctrine traite les objets PHP

Pour enregistrer des données dans la BD il faut préparer les objets contenant ces données la

La méthode `persist()` de l'EntityManager permet d'associer les objets à persister avec Doctrine

Afin d'exécuter les requêtes sur la BD (enregistrer les données dans la base) il faut utiliser la méthode `flush()`

L'utilisation de flush permet de profiter de la force de Doctrine qui utilise les Transactions

La persistance agit de la même façon avec l'ajout (insert) ou la mise à jour (update)

Le service EntityManager

Mise à Jour des données

```
public function AddUpdateAction($id)
{
    // on récupère notre entity manager
    $em = $this->getDoctrine()->getManager();
    $repo = $em->getRepository('Rt4AsBundle:Etudiant');
    // On prépare un nouvel étudiant
    $etudiant = new Etudiant();
    $etudiant->setNom('nouvel etudiant')
        ->setCin('12345678')
        ->setDateNaissance(new \DateTime())
        ->setPrenom('mon prenom')
        ->setNumEtudiant(1234);
    // On l'associe à Doctrine en le persistant
    $em->persist($etudiant);
    //on récupère une entité qui est automatiquement associé à Doctrine plus besoin de la persister
    $etud = $repo->find($id);
    if ($etud)
    {
        $etud->setCin(2782);
    }
    $em->flush();
    //on récupère l'ensemble des entités et on le transmet à la page d'affichage
    $etudiants = $em->getRepository("Rt4AsBundle:Etudiant")->findAll();
    return $this->render('Rt4AsBundle:Default:liester.html.twig', array(
        'etudiants' => $etudiants,
    ));
}
```

Enregistrement et mise
à jour des données
(Exemple)

Le service EntityManager

Suppression d'une entité

Suppression d'une entité

La méthode `remove()` permet de supprimer une entité

Exemple

```
public function DeleteAction($id)
{
    // on récupère notre entity manager et notre repository
    $em = $this->getDoctrine()->getManager();
    $repo = $em->getRepository('Rt4AsBundle:Etudiant');
    //on récupère une entité qui est automatiquement associé à Doctrine plus besoin de la persister
    $etud = $repo->find($id);
    if ($etud)
    {
        $em->remove($etud);
        $em->flush();
    }
    //on recupère l'ensemble des entités et on le transmet à la page d'affichage
    $etudiants = $em->getRepository("Rt4AsBundle:Etudiant")->findAll();
    return $this->render('Rt4AsBundle:Default:lister.html.twig', array(
        'etudiants' => $etudiants,
    ));
}
```

Gestion des relations entre les entités (1)

Les types de relation

Les entités de la BD présentent des relations d'association :

- A **OneToOne** B : à une entité A on associe une entité de B et inversement
- A **ManyToOne** B : à une entité B on associe plusieurs entité de A et à une entité de A on associe une entité de B
- A **ManyToMany** B : à une entité de A on associe plusieurs entité de B et inversement

Gestion des relations entre les entités (2)

Relation unidirectionnelle et bidirectionnelle

La notion de navigabilité de UML est la source de la notion de relation unidirectionnelle ou bidirectionnelle

Une relation est dite navigable dans les deux sens si les deux entité doivent avoir une trace de la relation.

Exemple : Supposons que nous avons les deux classes CandidatPresidentielle et Electeur.

L'électeur doit savoir à qui il a voté donc il doit sauvegarder cette information par contre le candidat pour cause d'anonymat de vote ne doit pas connaître les personnes qui ont voté pour lui.

On aura donc un attribut Candidat dans la table Electeur mais pas de collection ou tableau nommé électeur dans la table CandidatPresidentielle. Ici on a une relation **unidirectionnelle**.

Gestion des relations entre les entités (3)

OneToOne unidirectionnelle

- Relation **unidirectionnelle** puisque Media ne référence pas Etudiant

```
/**Entity **//  
Class Etudiant  
{  
// ...  
/**  
 * @ORM\OneToOne(targetEntity=  
targetEntity="Aymen\EntityRelationBundle\Entity\Media"  
)  
 */  
    private $media;  
// ...  
}  
/**Entity **//  
Class Etudiant  
{  
// ...  
}
```


Gestion des relations entre les entités (4)

OneToOne Bidirectionnelle

- Si nous voulons qu'à partir du media on peut directement savoir à quel étudiant il appartient nous devons faire une relation bidirectionnelle
- Media aussi doit référencer Etudiant

```
/**Entity **//
Class Etudiant
{
// ...
/**
 * @ORM\OneToOne(targetEntity=
"Aymen\EntityRelationBundle\Entity\Media")
 */
private $media;
// ...
}
/**Entity **//
Class Media
{
/**
 * @ORM\OneToOne(targetEntity=
"Aymen\EntityRelationBundle\Entity\Etudiant",mappedBy="media")
 */
private $etudiant;
// ...
}
```

Gestion des relations entre les entités (5)

ManyToOne Unidirectionnelle

- Relation **unidirectionnelle** puisque Section ne référence pas Etudiant

```
/**Entity **//  
Class Etudiant  
{  
  // ...  
  /**  
   * @ORM\ManyToOne(targetEntity="Aymen\EntityRelationBundle\Entity\Section")  
   */  
  private $section;  
  // ...  
}  
/**Entity **//  
Class Section  
{  
  // ...  
}
```

Gestion des relations entre les entités (6)

OneToMany Bidirectionnelle

- Si nous voulons connaître dans l'objet section l'ensemble des étudiants qui lui sont affectés alors on doit avoir une relation bidirectionnelle
- Section aussi doit référencer Etudiant
- On aura une relation **OneToMany** côté Section puisqu'à « **One** » Section on a « **Many** » Etudiants.
- On doit ajouter l'attribut **mappedBy** côté **OneToMany** et **inversedBy** côté **ManyToOne**
- On doit spécifier dans le **constructeur** du **OneToMany** que l'attribut mappé est de type **ArrayCollection** en l'instanciant

```
/**Entity **//
Class Section
{
// ...
/**
 * @ORM\OneToMany(targetEntity=
"Aymen\EntityRelationBundle\Entity\Etudiant", mappedBy="section")
 */
private $etudiants;
// ...
public function __construct() {
$this->etudiants = new ArrayCollection ();
}
}/**Entity **//
Class Etudiant
{ //...
/**
 * @ORM\ManyToOne(targetEntity=
"Aymen\EntityRelationBundle\Entity\Section", inversedBy="etudiants")
 */
private $section;
// ...
}
```

Gestion des relations entre les entités (7)

ManyToMany

- Relation **unidirectionnelle** puisque Cours ne référence pas Prof
- Ici on peut savoir quels sont les cours de chaque étudiant mais pas l'inverse (on peut l'extraire via une requête)

```
/**Entity **/  
Class Matiere  
{  
  // ...  
}  
/**Prof**/  
Class Etudiant  
{ //...  
  /**  
    * @ORM\ManyToMany(targetEntity=  
    "Aymen\EntityRelationBundle\Entity\Matiere")  
    */  
    private $cours ;  
  
    / ...  
  /**  
    * Constructor  
    */  
  public function __construct()  
  {  
    $this->matieres = new  
    \Doctrine\Common\Collections\ArrayCollection();  
  }  
}
```

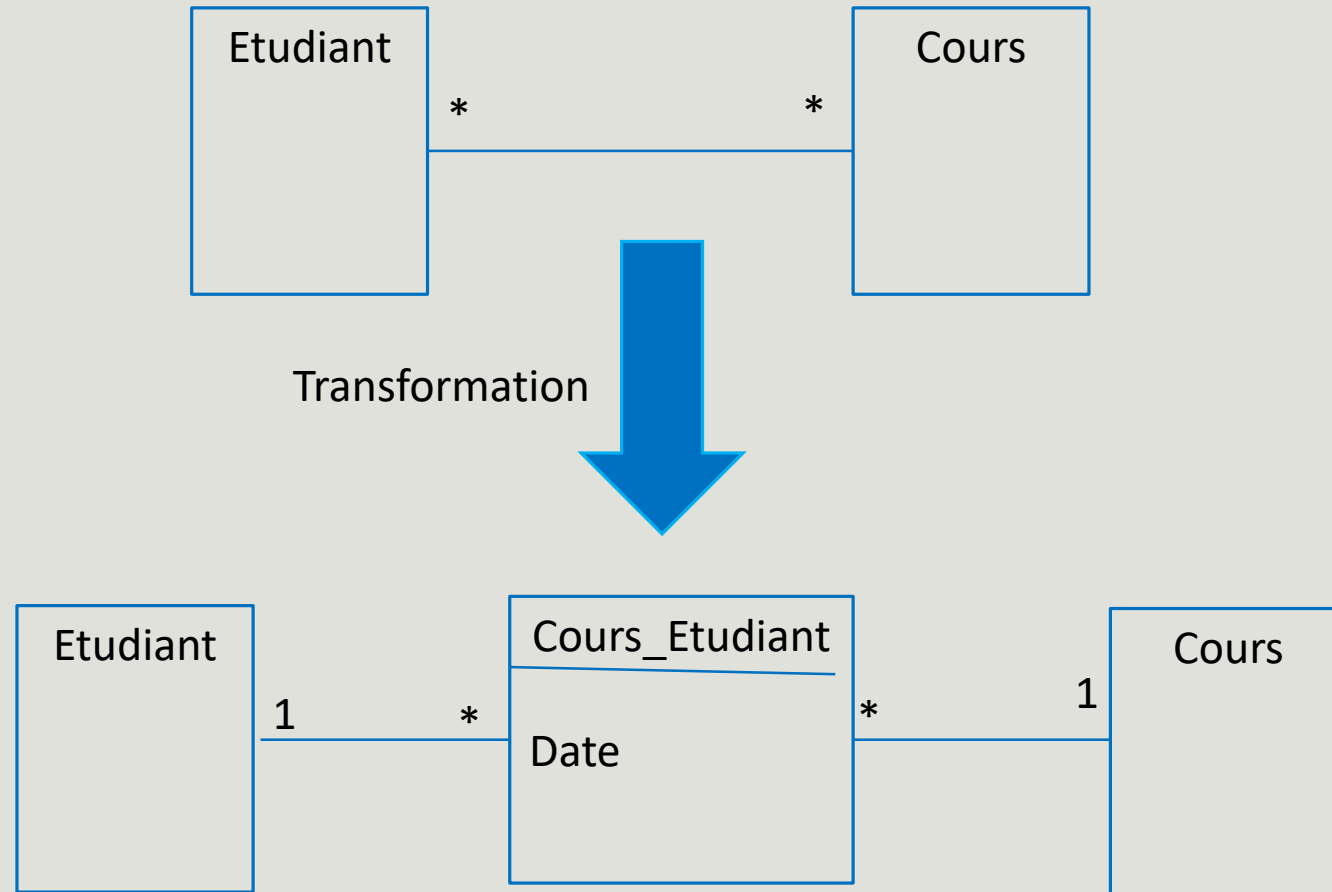
Gestion des relations entre les entités (8)

ManyToMany Bidirectionnelle

- On doit spécifier dans les deux constructeurs que l'attribut mappé est de type `ArrayCollection` en l'instanciant
- Même chose que la bidirectionnelle `OneToMany` en ajoutant les `mappedBy` et `inversedBy`
- Cette relation peut être traduite à deux relation `OneToMany/ManyToOne` entre les 3 classes participantes.

Gestion des relations entre les entités (9)

Cas particulier ManyToMany



Détour : les Traits

- Servent à externaliser du code redondant dans plusieurs classes différentes.
- Pourquoi les traits et non l'héritage ? Parce que PHP ne supporte pas l'héritage multiple.
- Non instanciable
- Un trait peut contenir des méthodes et des attributs
- Syntaxe :

```
Trait nomTrait{  
    public $x;  
    Function fct1(){  
    }  
    Function fct2(){  
    }  
}
```

Les événements de Doctrine (1)

➤ Appelé aussi les callbacks du cycle de vie, ce sont des méthodes à exécuter par doctrine dans moments et dépendant d'événement précis :

- PrePersist
- PostPersist (s'exécute après le `$em->flush()` non après `$em->persist()`)
- PreUpdate
- PostUpdate
- PreRemove
- PostRemove

Afin d'informer doctrine qu'une entité contient des callbacks nous devons utiliser l'annotation **HAasLifecycleCallbacks**

```
/**
 * Transfer
 *
 * @ORM\Table(name="transfer")
 *
 * @ORM\Entity(repositoryClass="AppBundle\
Repository\TransferRepository")
 * @ORM\HasLifecycleCallbacks()
 */
class Transfer
```

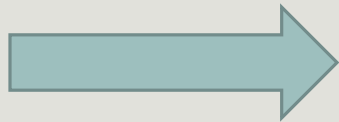

Les événements de Doctrine (2)

- Pour informer Doctrine de l'existence d'un événement on utilise maintenant l'annotation sur l'action à réaliser.

```
/**
 * @PrePersist()
 */
public function onPersist() {
    $this->createdAt = new \DateTime('NOW');
}
```

Les traits et Les événements de Doctrine

- Imaginons maintenant que nous voulons « sécuriser plusieurs de nos entités » et d'avoir un peu d'historique. L'idée est d'avoir deux attributs qui sont `createdAt` et `modifiedAt` pour avoir toujours une idée sur la création de notre entité et de sa dernière modification.
- L'idée est de créer pour chacune des entités à suivre des lifecycle callback qui vont mettre à jour ces deux attributs lors de la création (`prePersist`) et la modification (`preUpdate`).
- Est-ce normal de le refaire pour toutes les entités ?
- Si la réponse est non, que faire alors ?



Les traits