esprit
Se former autrement
HONORIS UNITED UNIVERSITIES

L'ORM (Object-Relational Mapping) : Doctrine

UP Web

AU: 2022/2023

United Nations Educational, Scientific and Cultural Organization
UNESCO Chair "Project-based learning" ESPRIT School of engineering, Tunisia
EUR-ACE® Délivrée par la Commission des Titres d'Ingénieur
CONFÉRENCE DES GRANDES ÉCOLES
cdio CONCEIVE DESIGN IMPLEMENT OPERATE

Plan

1. Introduction
2. Doctrine2
3. Les entités
4. La Migration
5. Les formulaires
6. Entity Manager: Manipuler les entités avec Doctrine2
7. Les relations entres entités avec Doctrine2

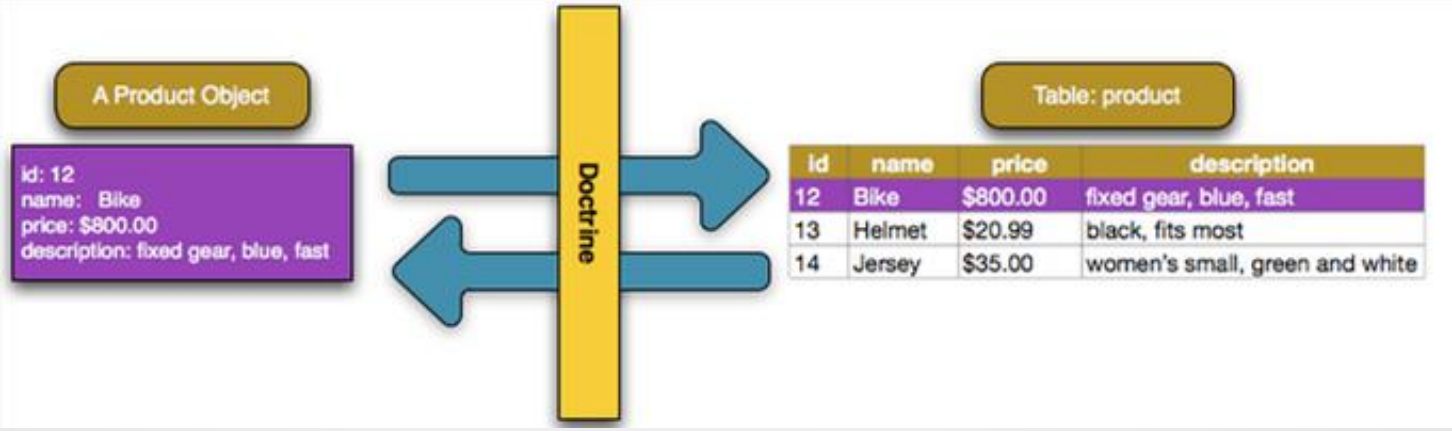
Introduction

- La programmation Orientée Objet, utilisant une base de données relationnelle, nécessite de convertir les données relationnelles en objets et vice-versa.

- Persistance d'objets métiers : les objets modélisés dans les applications sont associées à des données stockées dans les SGBDR

Object-Relational Mapping (ORM)

- C’est une couche d’abstraction à la base de données.
- ORM fait la relation entre les données orientées objet et les données relationnelles.



Object-Relational Mapping

Avantages:

- Simplifie l'accès aux données
- Facilite le changement de SGBDR
- une indépendance du code vis-à-vis du SGBDR utilisé

Object-Relational Mapping

- Les ORM les plus connus:

En Java: - Hibernate

- JPA (Java Persistance API)

- SimpleORM

En .NET: - Nhibernate

- Entity Framework

Quel choix pour PHP:

- **Doctrine**

- Propel

- RedBean

Doctrine

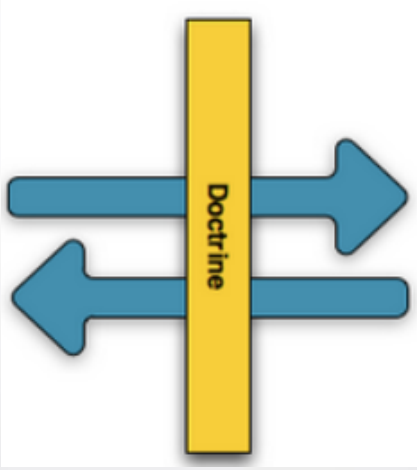
- C’est un ORM pour PHP
- Logiciel open source
- Dernière version stable: **2.13.2**
- Intégré dans différents Frameworks:
  - **Symfony**,
  - Zend Framework
  - CodeIgniter

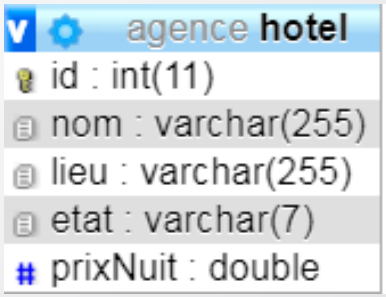
Doctrine - Caractéristiques

Une classe qui correspond à chaque table

Une classe = une « **Entité** »

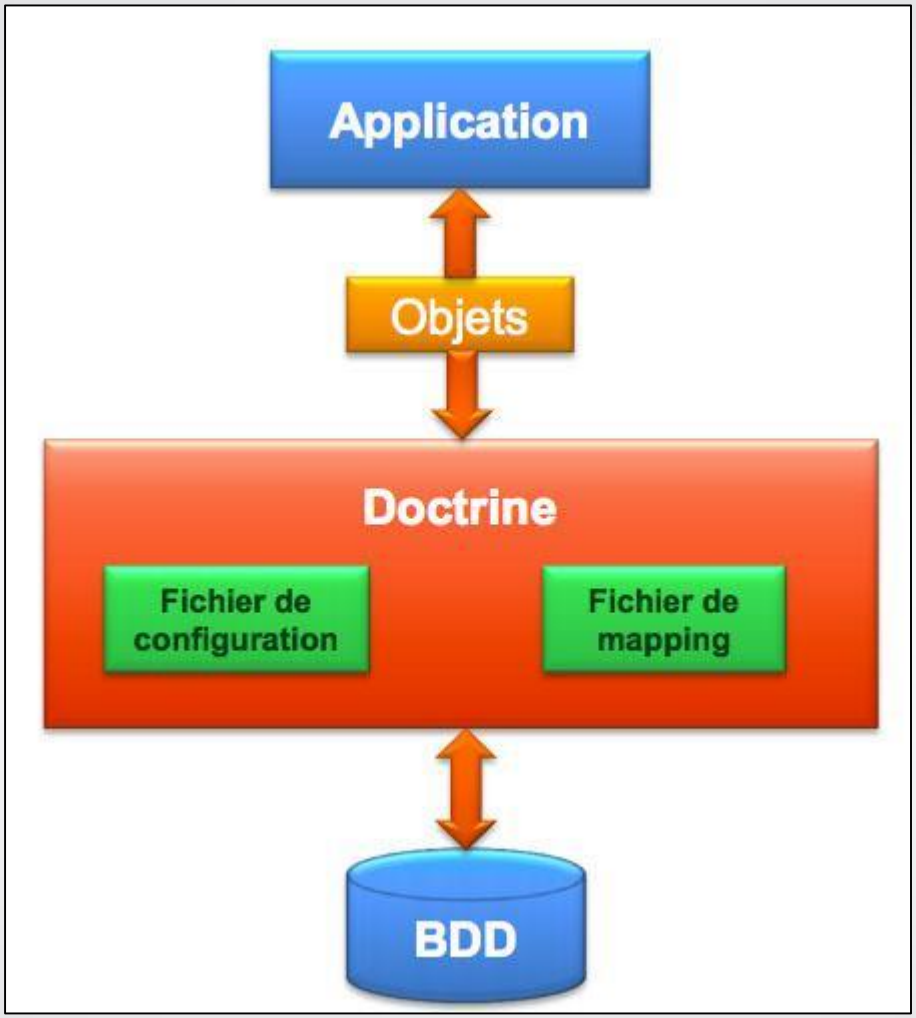
```php
class hotel
{

    private $id;


    private $nom;


    private $lieu;


    private $etat;

    public function getId()
    {
        return $this->id;
    }
    // .........
```





esprit

Doctrine – Architecture Technique



Architecture Technique de Doctrine

Doctrine - Caractéristiques

On a deux méthodes pour le Mapping:

- Fichier de mapping YAML, XML.
- Directement dans la classe via des annotations

Configuration de la base de données

Configurer la base de données de l'application dans le fichier **.env**

```
28 # IMPORTANT: You MUST configure your server version, either here or in config/packages/doctrine
29 #
30 # DATABASE_URL="sqlite:///%kernel.project_dir%/var/data.db"
31 # DATABASE_URL="mysql://app:!ChangeMe!@127.0.0.1:3306/app?serverVersion=8&charset=utf8mb4"
32 DATABASE_URL="postgresql://app:!ChangeMe!@127.0.0.1:5432/app?serverVersion=14&charset=utf8"
```

Nom de l'utilisateur par défaut « root »

Mot de passe de la base de données

Nom de la base de données

Exemple après les changements :

```
31 DATABASE_URL="mysql://root:@127.0.0.1:3306/3A30?"
```

Création de la base de données

La commande suivante permet de créer une base de données :

php bin/console doctrine:database:create
Ou
Symfony console doctrine:database:create

=>Une base de données avec les propriétés mentionnées dans **env** sera automatiquement générée

Les entités

Il existe deux méthodes pour générer les entités :

1. Méthode manuelle (non recommandée)

- Créer la classe
- Ajouter le mapping
- Ajouter les getters et les setters (manuellement ou en utilisant la commande suivante :

➔ **php bin/console make:entity --regenerate App**

➔ **symfony console make:entity --regenerate**

Les entités

2. Méthode en utilisant le bundle maker

- Ajouter une entité en lançant la commande suivante :

  ➔ **php bin/console make:entity**

  ➔ **symfony console make:entity**

- Ajouter les attributs et les paramètres

Les entités

Configuration de l'entité:

- **#[ORM\Id]:** spécifie la clé primaire
- **#[ORM\GeneratedValue]:** auto-incrémentée l'ID
- **#[ORM\Column]:** s'applique sur un attribut de la classe et permet de définir les caractéristiques de la colonne concernée (nom , taille , types, etc.)

```php
#[ORM\Id]
#[ORM\GeneratedValue]
#[ORM\Column]
private ?int $id = null;
```

La Migration

La migration permet de tracker les différents changements et évolutions de votre base de données

Une image = Une évolution de la base de donnée

La commande ci-dessous nous permet de créer un fichier de migration:

➔ **php bin/console make:migration**
➔ **symfony console make:migration**

La commande ci-dessous nous permet de lancer une migration:

➔ **php bin/console doctrine:migrations:migrate**
➔ **symfony console doctrine:migrations:migrate**

Le fichier de migration est générée en se basant sur la date et l'heure actuelle sous le nom « **YYYYMMDDHHMMSS »**

Le nom de fichier de migration est un **timestamp** (La valeur représentant la date et l'heure) qui sera stocké dans la base dans une table **doctrine_migration_versions.**

La Migration

Chaque fichier de migration possède trois méthodes:

- La méthode **getDescription()** :permet de décrire la migration
- La méthode **up():** est exécutée lorsqu'on applique la migration en utilisant la commande **"php bin/console doctrine:migrations:migrate"**

```
$ php bin/console doctrine:migrations:migrate

 WARNING! You are about to execute a database migration that could result in schema changes
 and data loss. Are you sure you wish to continue? (yes/no) [yes]:
 >

[notice] Migrating up to DoctrineMigrations\Version20210122210730
[notice] finished in 217.6ms, used 18M memory, 1 migrations executed, 1 sql queries
```

- La méthode **down()** :est exécutée lorsqu'on annule la migration en utilisant la commande "**php bin/console doctrine:migrations:migrate prev**"

```
$ php bin/console doctrine:migrations:migrate prev

 WARNING! You are about to execute a database migration that could result in schema changes
 and data loss. Are you sure you wish to continue? (yes/no) [yes]:
 >

[notice] Migrating down to DoctrineMigrations\Version20210117184800
[notice] finished in 447.7ms, used 18M memory, 1 migrations executed, 1 sql queries
```

La Migration

doctrine:migrations:current	Afficher la version actuelle
doctrine:migrations:execute version	Exécuter une seule version de migration
doctrine:migrations:generate	Créer un fichier de migration vide
doctrine:migrations:latest	Afficher la dernière version migration
doctrine:migrations:migrate	Exécuter toutes les versions de migrations non exécutées
doctrine:migrations:status	Afficher l'état d'un ensemble de migrations
doctrine:migrations:up-to-date	Nous indiquer si le schéma est à jour
doctrine:migrations:version version -- add/delete	Ajouter ou supprimer manuellement les versions de migration de la table des versions.
doctrine:migrations:sync-metadata-storage	Vérifier si le stockage des métadonnées est à jour
doctrine:migrations:list	Afficher la liste de toutes les migrations disponibles et leur état
doctrine:migrations:migrate next	Executer la méthode up de la premiere migration générée et non exécutée (une seule)
doctrine:migrations:migrate prev	Exécuter la méthode down de la dernière migration

(*) version: namespace\version
Exemple: DoctrineMigrations\Version20201013084414

Le Repository

- Pour chaque entité, il existe un Repository (Exemple: StudentRepository est associé à l'entité Student)
- Un Repository centralise tout ce qui touche à la récupération des entités
- C'est une classe PHP qui contient les méthodes de récupération de données relatives aux entités.
- Un Repository utilise plusieurs types d'entités, dans le cas d'une jointure par exemple.
- L'appel de la classe Repository se fait dans la classe Entity:

```
#[ORM\Entity(repositoryClass: StudentRepository::class)]
class Student
```

Le Repository

Il existe 2 façons pour récupérer le repository

Les repositories héritent de la classe

Doctrine\Persistence\ManagerRegistry

Il faut injecter la classe ManagerRegistry dans la méthode

public function listStudent(**ManagerRegistry $doctrine**)

$repository=$doctrine->getRepository(NomClasse::class);

En utilisant l'injection de dépendance

public function listStudent(**StudentRepository $repository**)

Il existe 3 façons pour récupérer les *objets* :

- les méthodes de récupération de base: findAll(),findBy(),find($id)
- les méthodes magiques findByX(), findOneByX()
- les méthodes de récupération personnalisées: DQL/QueryBuilder

Les méthodes de récupération de base

Méthode	Description
find($id)	Trouver un objet à partir son ***id***
findAll()	Trouver tous les objets
findBy()	Trouver plusieurs objets à partir d'un ou plusieurs d'attributs
findOneBy()	même principe que findBy mais elle retourne un seul objet

Les méthodes de récupération de base

La méthode: *findAll()*

- **Rôle :** retourne tous les objets ou les enregistrements contenue dans la base de données. Le format de retour est un tableau PHP normal (un array).

- **Exemple :**

```
$students=$doctrine->
getRepository(Student::class)->findAll();
```

Les méthodes de récupération de base

La méthode: *find($id)*

- **Rôle :** retourne l'objet ou l'enregistrement qui correspond à la clé primaire passée en argument. Généralement cette clé est l'id.

- **Exemple :**

```
$student=$doctrine->
getRepository(Student::class)->find(2);
```

Les méthodes de récupération de base

La méthode: *findBy()*

- **Rôle:** permet de retourner une liste d'objets en appliquant un filtre pour ne retourner que les entités correspondant à un ou plusieurs critère(s).
- **Syntaxe:**

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

- **Exemple :**

```
$student=$doctrine->getRepository(Student::class)-
>findBy(array('lastname' => 'foulen','firstname' => 'ben
foulen'),array('lastname' => 'desc'),10, 0);
```

Les méthodes de récupération de base

La méthode: *findOneBy()*

- **Rôle:** Même principe que findBy() mais en retournant un seul objet ce qui élimine automatiquement les paramètres d'ordre de limite et d'offset
- **Syntaxe:**

```
findOneBy(array $criteria, array $orderBy = null)
```

- **Exemple :**

```
$student=$doctrine-> ->getRepository(Student::class)-
>findOneBy(array('lastname' => 'foulen','firstname' =>
'ben foulen'));
```

Les méthodes de récupération magiques

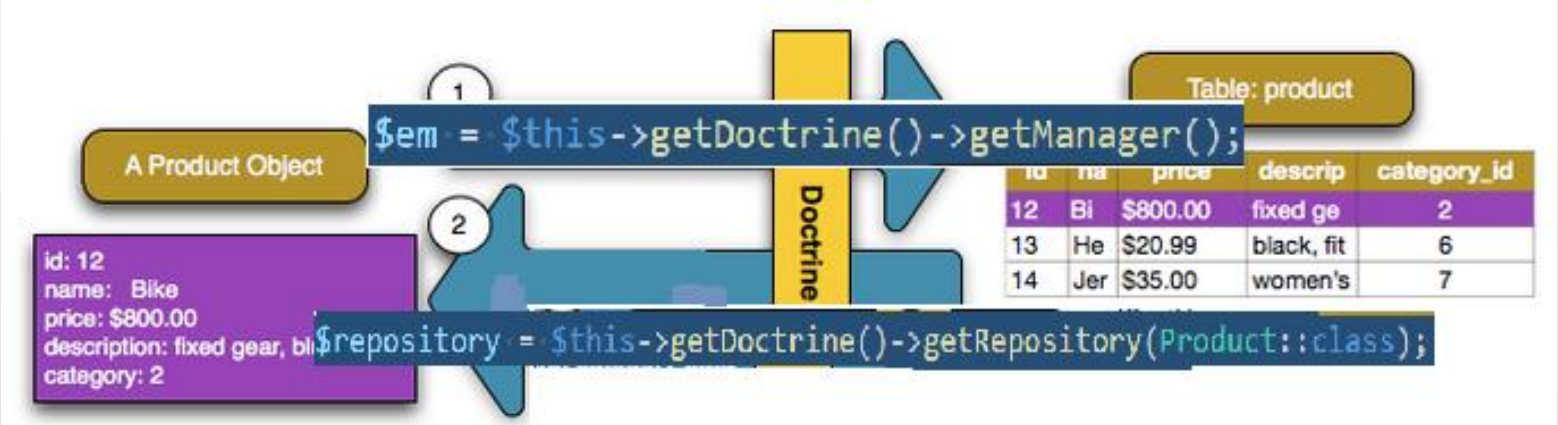
Méthode	Description
findByX($valeur) en remplaçant X par un attribut de l'objet Exemple: findBynsc('05896542') findByNom('Club Info')	Similaire à findBy() avec un seul critère, celui du nom de la méthode
findOneByX($valeur) en remplaçant X par un attribut de l'objet Exemple: findOneBynsc('05896542')	Similaire à findOneBy() mais en retournant un seul objet

Entity Manager

- EM est un gestionnaire d'entités: le chef d'orchestre de l'ORM Doctrine
- EM est placée entre les objets (entités) et les tables de la base de données.
- EM permet l'insertion, la mise à jour et la suppression des données de la base de données

Tout d'abord, On doit récupérer l'entity manager

$em= $doctrine->getManager()



Manipulation des entités avec Doctrine2

- La méthode **persist()** :Utilisée pour l'ajout d'un nouvel objet afin d'informer Doctrine que l'on veut ajouter cet objet dans la base de données.

➢ **$em->persist($object)**

- La méthode **flush()** permet d'envoyer tout ce qui a été persisté avant à la base de données afin d'exécuter la requête. Cette méthode est utilisée pour l'ajout, la modification et la suppression

➢ **$em->flush()**

Manipulation les entités avec Doctrine2

- La méthode **remove()** indique à Doctrine d'exécuter la requête de suppression de l'entité en argument de la base de données
  - **$em->remove($object)**
- La méthode **clear()** permet d'annuler tous les persist
  - **$em->clear()**
- La méthode **detach()** permet d'annuler le persist effectué sur l'entité en argument
  - **$em->detach($object)**

Création d'un Formulaire

- Un formulaire Symfony est l'image d'un objet existant

- Un formulaire sert à alimenter un objet.
- Un formulaire sert à récupérer des informations indépendantes de n'importe quel objet.
- Pour installer les dépendances du formulaire, il faut exécuter cette commande: **composer require symfony/form**

Création d'un Formulaire

Principe

On peut définir un formulaire soit:

1. dans le contrôleur
2. dans un autre fichier qui sera appelé par le contrôleur

Création d'un Formulaire

Méthode 1: Création d'un formulaire dans le contrôleur

Pour indiquer les champs à ajouter au formulaire on utilise la méthode **add** du **FormBuilder**

Cette méthode contient 3 paramètres :
1) le **nom** du champ dans le formulaire
2) le **type** du champ
3) Un tableau (**array,[])** qui contient des options spécifiques au type du champ

```
public function AddFormClassRoomBuilder()
{
    $classroom = new ClassRoom();
    $classroom->setLevel('Write a level');
    $classroom->setBranch('Write a branch');

    $form = $this->createFormBuilder($classroom)
        ->add('level', TextType::class)
        ->add('branch', TextType::class)
        ->add('save', SubmitType::class, ['label' => 'Create Class'])
```

Création d'un Formulaire

Problématique:

Comment alléger le contrôleur ?

Solution:

Générer une classe qui s'occupe de la construction d'un formulaire

Comment?

utilisation de la commande:

php bin/console make:form FormName

symfony console make:form FormName

Création d'un Formulaire

Méthode 2: Utilisation de la commande

La commande suivante nous permet de créer un Formulaire :

php bin/console make:form FormName

symfony console make:form FormName

Maker vous demandera si votre formulaire est associé à une entité ou non. Répondez selon votre besoin.

Un objet dont le nom est **« FormNameType »** sera automatiquement créé dans le dossier **src/Form** et qui contient une fonction **buildForm**

```php
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('level')
        ->add('branch')
    ;
}
```

Création d'un Formulaire

Récupération du formulaire dans le contrôleur

La récupération du formulaire au niveau des contrôleurs devient beaucoup plus facile :

```
public function AddFormClassRoomMaker(Request $request)
{
    $classroom = new ClassRoom();

    $form = $this->createForm(ClassRoomType::class, $classroom);
```

Le second paramètre « **$nomClass**» n'est pas obligatoire

Création d'un Formulaire

Types dans le formulaire:

Il existe différents types proposés par Symfony pour les champs d'un formulaire dont chaque champ possède un nom, un type et des options.

Text Fields

- TextType
- TextareaType
- EmailType
- IntegerType
- MoneyType
- NumberType
- PasswordType
- PercentType
- SearchType
- UrlType
- RangeType
- TelType
- ColorType

Choice Fields

- ChoiceType
- EntityType
- CountryType
- LanguageType
- LocaleType
- TimezoneType
- CurrencyType

Date and Time Fields

- DateType
- DateIntervalType
- DateTimeType
- TimeType
- BirthdayType
- WeekType

Field Groups

- CollectionType
- RepeatedType

Other Fields

- CheckboxType
- FileType
- RadioType

Buttons

- ButtonType
- ResetType
- SubmitType

Hidden Fields

- HiddenType

Base Fields

- FormType

Création d'un Formulaire

Champ choice particulier:

Les choices Type seront chargés à partir des éléments d'une entité Doctrine. Il existe différentes façons d'implémenter ces choices type. Les plus importants sont les deux attributs expanded et multiple qui construisent la façon d'afficher cette liste de choix.

Balise HTML	expanded	multiple
Liste déroulante	false	false
Liste déroulante (avec attribut `multiple`)	false	true
Boutons radio	true	false
Cases à cocher	true	true

```
->add('Classroom', EntityType::class, [
    'class'=>Classroom::class,
    'choice_label'=>'name',
    'multiple'=>false,
    'expanded'=>false,
    ])
```

Création d'un Formulaire

L'envoi du formulaire à la page twig

➔ return $this->render('formation/.html.twig', ['formA' => $form]);

Affichage du formulaire dans TWIG

➢ Afficher la totalité du formulaire avec la méthode form() qui affiche le formulaire sans aucune mise en forme

{{ form(nomDuFormulaire) }}

```
{{ form(formA) }}
```

Création d'un Formulaire

Afficher les composants du formulaire séparément un à un d'une façon personnalisée

{{ form(nomDuFormulaire) }}

=

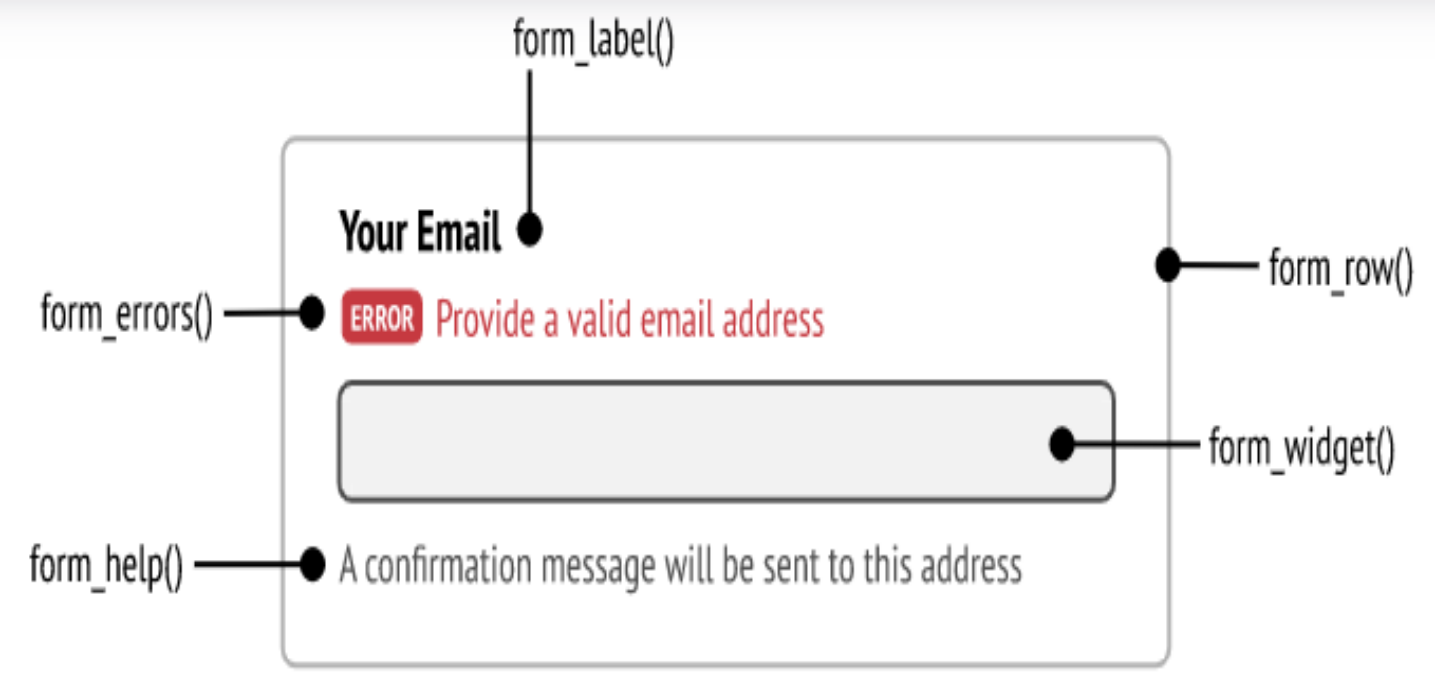
Form_start(): est l'équivalent de <form> en HTML
Form_errors(): affiche les erreurs relatives au champ passé en argument
Form_label(): affiche le label HTML d'un élément du formulaire
Form_widget(): affiche le champ du formulaire
Form_help(): affiche l'aide relatives au champ donné
Form_row(): est l'équivalent de form_label + form_errors + form_widget

```
{{ form(formA) }}
```

```
{{ form_start(formA) }}
    {{ form_label(formA.email,"Your Email")}}
    {{ form_errors(formA.email) }}
    {{ form_widget(formA.email)}}
    {{ form_help(formA.email)}}
{{ form_end(formA) }}
```

```
{{ form_row(formA.email) }}
```

Création d'un Formulaire



Les relations entre les entités

Les types de relation possibles

Une relation (ou une association) peut être:

- ***Unidirectionnelle :*** Seules les instances de l'une des entités de l'association peuvent retrouver les instances de l'entité partenaire.

  ⇒ **Par exemple :** un utilisateur peut obtenir la liste de ses adresses connues, par contre il n'est pas possible de retrouver un utilisateur à partir d'une adresse.

Les relations entre les entités

Limites:

- ❏ Les relations sont unidirectionnelles
- ❏ On peut faire $student->getProjects()
- ❏ Mais on ne peut pas faire $project->getStudent()

Solution:

- ❏ Rendre les relations bidirectionnelles

Les relations entre les entités

Les types de relation possibles

➢ ***Bidirectionnelle:*** Les instances de l'une ou de l'autre des entités de l'association peuvent retrouver les instances de l'entité partenaire.

⇒ **Par exemple :** un utilisateur peut obtenir la liste des commandes qu'il a effectué et on peut retrouver un utilisateur à partir d'une commande.

Les relations entre les entités

One To One ,bidirectionnel

Exemple : un étudiant possède sa propre carte d'étudiant.

⇒ ***mappedBy*** fait référence à l'attribut ***card*** dans la classe ***Student***

```php
class StudentCard
{
    #[ORM\OneToOne(mappedBy: 'card', cascade: ['persist', 'remove'])]
    private ?Student $student = null;
```

⇒ ***inversedBy*** fait référence à l'attribut ***student*** dans la classe ***Studentcard***

```php
class Student
{
    #[ORM\OneToOne(inversedBy: 'student', cascade: ['persist', 'remove'])]
    #[ORM\JoinColumn(nullable: false)]
    private ?StudentCard $card = null;
```

Les relations entre les entités

One To Many ,bidirectionnel

Exemple : Une classe contient plusieurs étudiants.

```php
class Classroom
{
    /**
     * @ORM\OneToMany(targetEntity="App\Entity\Student"
     * mappedBy="classroom")
     */
    private $students;
```

- ***targetEntity :*** namespace complet vers l'entité liée.
- ***mappedBy :*** il s'agit de l'attribut de l'entité cible qui illustre la relation entre les deux entités

```php
class Student
{
    /**
     * @ORM\ManyToOne(targetEntity="App\Entity\Classroom"
     *      inversedBy="students")
     */
    private $classroom;
```

NB : Obligatoirement dans l'entité *target* il faut avoir une définition d'attribut avec le mot clé *ManyToOne*

Les relations entre les entités

Many To Many ,bidirectionnelle avec table de jointure

- On peut générer la relation entre les deux entités automatiquement:

  Prenons l'exemple d'une relation ***ManyToMany***

  ⇒ Plusieurs étudiants peuvent appartenir à plusieurs clubs.

- On doit tout d'abord modifier notre entité "**Student**" en tapant la commande suivante:

```
C:\wamp65\www\myproject-course>php bin/console make:entity

 Class name of the entity to create or update (e.g. TinyPopsicle):
 > Student

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

Les relations entre les entités

Many To Many ,bidirectionnelle avec table de jointure

- Maintenant il faut ajouter l'attribut, dans notre cas "**clubs**":

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

- Spécifier le nom de la classe avec laquelle est reliée qui "***Club***":

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

- Spécifier le type de cet attribut, tapez "**relation**" :

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

Les relations entre les entités

Many To Many ,bidirectionnelle avec table de jointure

- Par la suite , veuillez indiquer le type de relation : **ManyToMany**

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

Les relations entre les entités

Many To Many ,bidirectionnelle avec table de jointure

Student.php

```php
class Student
{
    /**
    *Many Students have many clubs
    * @ORM\ManyToMany (targetEntity="App\Entity\Club",inversedBy="students")
    * @ORM\JoinTable(
    *     joinColumns={@ORM\JoinColumn(onDelete="CASCADE")},
    *     inverseJoinColumns={@ORM\JoinColumn(onDelete="CASCADE")} * ) */
    private  $clubs;
```

Club.php

```php
class Club
{
    /**
     * Many Clubs can contains many Students
     * @ORM\ManyToMany (targetEntity="App\Entity\Student",mappedBy="clubs")
     */
    private $students;
    public function __construct(){
        $this->students=new ArrayCollection();
    }
```

Les relations entre les entités

Cascade

Dans doctrine2, toutes les opérations de cascade sont par défaut désactivées, c'est-à-dire qu'aucune opération ne sera cascadée.

Si une entité est supprimée ou modifiée, les entités avec lesquelles elle était en relation ne seront pas supprimées ou modifiées dans la base de données.

Pour réaliser la suppression ou la modification en cascade, il faut rajouter l'option **cascade={"action"}** dans le mapping de l'entité.

Les relations entre les entités

Les actions en cascade

persist: Si l'entité est sauvegardée, faire de même avec les entités associées
remove: Si l'entité est supprimée, faire de même avec les entités associées.
merge: Cascades fusionne les opérations avec les entités associées.
detach: Cascades détache les opérations aux entités associées.
refresh: les opérations d'actualisation des cascades vers les entités associées.
all: les cascades persistent, suppriment, fusionnent, actualisent et détachent les opérations aux entités associée

Les relations entre les entités

Exemples

```
/**
 * @ORM\OneToMany(targetEntity=Student::class,
 * mappedBy="Classroom",
 * cascade={"all"},
 * )
 */
private $students;
```

```
/**
 * @ORM\OneToMany(targetEntity=Student::class,
 * mappedBy="Classroom",
 * cascade={"remove"},
 * )
 */
private $students;
```

```
/**
 * @ORM\OneToMany(targetEntity=Student::class,
 * mappedBy="Classroom",
 * cascade={"persist"}
 * )
 */
private $students;
```

Les relations entre les entités

Cascade

Une style de cascade spécial, **delete-orphan**, s'applique seulement aux associations un-vers-plusieurs si vous répondez par oui à la dernière question

```
Do you want to activate orphanRemoval on your relationship?
A Agences is "orphaned" when it is removed from its related Villes.
e.g. $villes->removeAgences($agences)

NOTE: If a Agences may *change* from one Villes to another, answer "no".

Do you want to automatically delete orphaned App\Entity\Agences objects (orphanRemoval ? (yes/no) [no]:
> yes
```

```
/**
 * @ORM\OneToMany(targetEntity=Student::class,
 * mappedBy="Classroom",
 * orphanRemoval=true)
 */
private $students;
```

Atelier 4

Références

- https://symfony.com/doc/master/bundles/DoctrineMigrationsBundle/index.htmlhttp://php.net/manual/fr/language.oop5.magic.php
- https://symfony.com/doc/current/doctrine.html#doctrine-queries