TP3 – Symfony 5 : Créer un catalogue en ligne

Nous allons développer un catalogue de vente de produits divers.

I. Les entités

❖ Créer un nouveau projet ***symfony*** appelé «***eshop*** »

```
C:\wamp64\www>composer create-project symfony/website-skeleton eshop
```

❖ Modifier le fichier ***.env*** pour configurer l'accès à la bdd (à compléter)

```
###> doctrine/doctrine-bundle ###
# Format described at http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html#connecting-using-a-url
# For an SQLite database, use: "sqlite:///%kernel.project_dir%/var/data.db"
# Configure your db driver and server_version in config/packages/doctrine.yaml
DATABASE_URL=mysql://:@127.0.0.1:3306/eshop
###< doctrine/doctrine-bundle ###
```

❖ Créer ensuite la bdd :

```
C:\wamp64\www\eshop>php bin/console doctrine:database:create
```

❖ **Générer** les deux **entités doctrine** suivantes :

```
C:\wamp64\www\eshop>php bin/console make:entity
```

Produit

- libelle string 50
- description text
- prix**HT** decimal Precision : 10 Scale: 3 (le nombre de chiffres après la virgule)

Categorie

- libelle string 50 UNIQUE
- description text

Une fois l'entité (la classe créée), nous allons demander à Doctrine de créer la table sous MySQL. En mode console saisir la commande suivante:

```
$ php bin/console make:migration
```

Si tout a fonctionné, vous devriez voir quelque chose comme ceci :

SUCCÈS !

Suivant : Consultez la nouvelle migration "src / Migrations / Version20190207231217.php"

Ensuite : Exécutez la migration avec php bin / console doctrine: migrations: migrate

Si vous ouvrez ce fichier, il contient le code SQL nécessaire pour mettre à jour votre base de données ! Pour exécuter ce SQL, exécutez vos migrations :

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

Cette commande exécute tous les fichiers de migration qui n'ont pas encore été exécutés sur votre base de données. Vous devez exécuter cette commande sur la production lors du déploiement pour maintenir votre base de données de production à jour.

On sait que :

- un produit est lié à une seule catégorie : on l'appellera «***propriétaire***»
- une catégorie est liée à plusieurs produits : on l'appellera «***inverse***»

Cette relation est appelée « ***ManyToOne***» : On a plusieurs (***many***) produits pour (***to***) une (***one***) catégorie.

La classe ***Produit*** doit avoir une nouvelle propriété qu'on appellera «***categorie***», elle pointera vers la classe «***Categorie***», et elle est ***obligatoire*** (donc non nulle)

```
php bin/console make:entity

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

 to stop adding fields):
> categorie

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

What class should this entity be related to?:
> Categorie

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

Is the Product.category property allowed to be null (nullable)? (yes/no) [yes]:
```

```
> no

Do you want to add a new property to Category so that you can access/update getProduits()? (yes/no)
[yes]:
> yes

New field name inside Category [produits]:
> produits

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

 to stop adding fields):
>
(press enter again to finish)
```

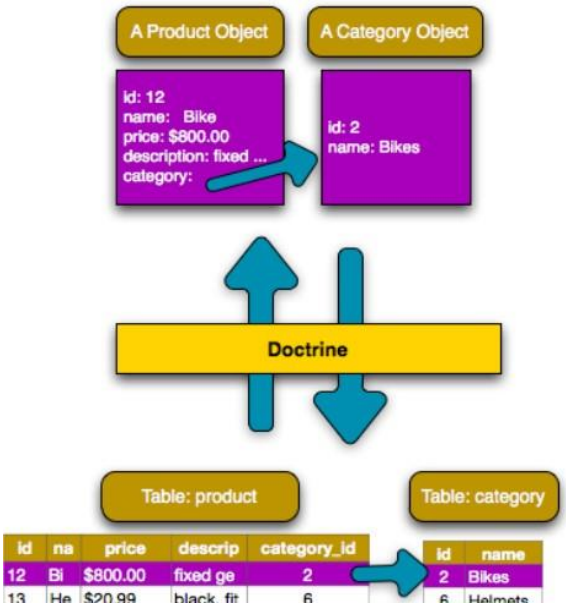
Categorie.php :

```
/**
     * @ORM\OneToMany(targetEntity="App\Entity\Produit", mappedBy="categorie")
*/
private $produits;
```

Produit.php :

```
/**
     * @ORM\ManyToOne(targetEntity="App\Entity\Categorie", inversedBy="produits")
* @ORM\JoinColumn(nullable=false)
     */
private $categorie;
```

Voici un schéma explicatif :



Vérifier via adminer (ou phpmyadmin) que la table « produit » a bien été mise à jour et qu'il y a une nouvelle clé étrangère dont le nom est composé de la classe cible (categorie) suivie d'un ***undescore (_)*** et son identifiant unique (id) :

Column	Type	Comment
	Auto Increment	
categorie_id	int(11)	
	(50)	
description	longtext	
prixHT	decimal(10,3)	

Indexes

PRIMARY	*id*
INDEX	*categorie_id*

Alter indexes

Foreign keys

Source	Target	ON DELETE	ON UPDATE	
categorie_id	categorie(*id*)	RESTRICT	RESTRICT	Alter

Pour info notons que le moteur (engine) utilisé est de type « **innoDB** », c'est à grâce à ce moteur que mySQL peut gérer les clés étrangères (ce qui n'est pas le cas via le moteur par défaut « myISAM »)

produit	InnoDB	utf8_un

II. Générateur de CRUD pour l'entité « Categorie»

On va maintenant générer les fichiers nécessaires à la gestion des catégories : lancer la commande suivant et suivre les étapes ci-dessous (**les questions** sont en gras, les réponses manuelles sont en couleur de fond)

```
$ php bin/console make:crud
```

```
The class name of the entity to create CRUD (e.g. GrumpyJellybean):
>Categorie
```

Cette commande vient de générer plusieurs fichiers :

- Plusieurs vues dans le dossier***/templates/categorie***
- Un controller appelé***/Src/Controller/CategorieController.php***
- La classe pour le formulaire***/Src/Form/ControllerType.php***

Tester toutes les pages (liste, ajout, modification et suppression) depuis l'url http://127.0.0.1:8000/categorie

ATTENTION: avant de continuer, ajouter les catégories suivantes : Animaux, Technologie, Nature

III. Générateur de CRUD pour l'entité «Produit»

Effectuer les mêmes étapes pour générer les CRUD de l'entité «Produit» Essayer de créer un nouveau produit, vous aurez un message d'erreur!!!

Le problème se trouve dans « ***ProduitType*** », ouvrez le code du fichier :

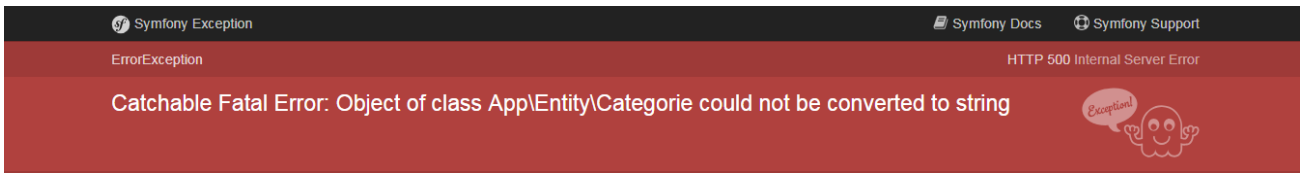
/src/Form/ProduitType.php

```
public function buildForm(FormBuilderInterface $builder, array $options)
    {
$builder
            ->add('libelle')
            ->add('description')
->add('prixHT')
->add('categorie')
        ;
    }
```

Ce code génère les champs du formulaire, cependant le type de chaque champ n'est pas indiqué, symfony essaie alors de convertir tout en champ de type texte (donc de type String), or rappelle ce qu'on a dans l'entité Produit :

```
….
/**
     * @ORM\ManyToOne(targetEntity="App\Entity\Categorie", inversedBy="produits")
     * @ORM\JoinColumn(nullable=false)
     */
private$categorie;
….
```

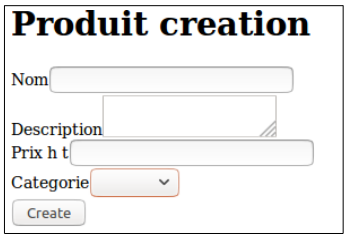
Donc «categorie» est une entité (donc un **objet)** qui est impossible à convertir en chaîne de caractère ! D'où me message d'erreur lors de l'ajout d'un nouveau produit :



Ouvrez le fichier ***/src/Entity/Categorie.php*** et ajoutez **à la fin** la méthode suivante :

```
public function __toString()
{
return $this->libelle;
}
```

Essayer maintenant d'ajouter un nouveau produit : ça marche :) et en plus on a une liste déroulante pour choisir la catégorie du produit :



Modifiez le code de cette vue comme index comme suit :

```
            <tr>
                <th>Id</th>
                <th>Nom</th>
                <th>Description</th>
                <th>Prix HT</th>
                <th>Catégorie</th>
                <th>Actions</th>
            </tr>
        </thead>
        <tbody>
        {% for produit in produits %}
            <tr>
                <td><a href="{{ path('catalogue_produit_sh
                <td>{{ produit.nom }}</td>
                <td>{{ produit.description }}</td>
                <td>{{ produit.prixHT }}</td>
                <td>{{ produit.categorie.libelle }}</td>
```

Actualiser la liste : on a bien la catégorie dans la liste.

IV. Appliquer Bootstrap & Mise enforme

Modifier ***/templates/base.html.twig*** comme suit :

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">

<title>{% block title %}Welcome!{% endblock %}</title>
        {% block stylesheets %}{% endblock %}
<link rel="icon"type="image/x-icon"href="{{ asset('favicon.ico') }}" />
<link rel="stylesheet"
href="//maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
</head>
<body>
<div class="container">
        {% block body %}{% endblock %}
</div>
        {% block javascripts %}{% endblock %}
</body>
</html>
```

Appliquer Bootstrap aux formulaires en modifiant le fichier ***config/packages/twig.yaml***

Chercher le bloc ‘twig’ ci-dessous et ajouter ceci :

```
twig:
    default_path: '%kernel.project_dir%/templates'
    debug: '%kernel.debug%'
    strict_variables: '%kernel.debug%'
    form_themes: ['bootstrap_4_layout.html.twig']
```

ATTENTION: Ajouter pour chaque catégorie 2 ou 3 produits.

V. Ajout d'un champ « image»

Ajouter à l'entité ***Categorie*** un attribut de type string (255 caractères) appelé «image». Ajouter aussi un champ «***image***» pour ***Produit***.

Générer les setters/getters via la commande

```
php bin/console make:entity --regenerate App
```

Dans la classe ***CategorieType*** ajouter le nouveau champ juste après le libellé. Idem pour ***ProduitType***.

ATTENTION : en pratique un champ image est de type «***FileType***» et se présente sous forme d'un bouton permettant de charger (***upload***) un fichier depuis la machine du client, mais pour simplifier le TP on va utiliser un champ texte où on indiquera l'URL d'une image.

Exp:

https://placeimg.com/200/150/**nature | tech | animals**

http://ipsumimage.appspot.com/140x100

http://via.placeholder.com/100x150?text=Hello

ATTENTION: Mettre à jour les produits et les catégories en définissant une image pour chaque catégorie et une image pour chaque produit. Donner des dimensions légèrement différentes (exp 200/150, 201/150, 200/151) afin d'avoir des images différentes.

VI. Front office : consulter les catégories

Créer un nouveau controller ***FrontController*** avec une méthode appelée ***indexAction*** :

- ***Route :*** /eshop/
- ***Nom de la route :*** eshop_index

Cette action récupère la liste des catégories et la passe à la vue «***front/index.html.twig***». Compléter son code:

```
/**
    * @Route("/eshop", name="eshop_index")
    */
   public function indexAction()
   {
      $em = $this->getDoctrine()->getManager();
      $repo = $em->getRepository(Categorie::class);

      $categories = $repo->findAll();
return $this->render('front/index.html.twig',['categories'=>$categories]);
  }
```

```
{% extends 'base.html.twig' %}
{% block body %}
<h1>Liste des catégories</h1>
<div class="row">
{% for cat in categories %}
<div class="col-sm-4">
<div class="thumbnail">
<img src="{{cat.image}}" alt="">
<div class="caption text-center">
<h3>{{cat.libelle}}</h3>
<a href="#" class="btn btn-primary">Produits</a>
</div>
</div>
</div>
{% endfor   %}
</div>
{% endblock %}
```

Le résultat doit être similaire à ceci :

VII. Front office : consulter les produits d'une catégorie

Le clic sur le bouton «***Produits***» (de l'écran précédent) doit ouvrir une nouvelle page qui affiche le nom de la catégorie tout en haut, sa description et juste au-dessous la liste de ses produits (sous forme d'image, titre et description) Voici le résultat si on clique sur la catégorie «Animals»

Voici l'action à compléter. ATTENTION : utiliser le ***ParamConverter*** afin de récupérer l'objet Categorie, **n'ajouter *aucune* ligne**.

```
/**
* @Route("/eshop/categorie/{id}", name=" eshop_categorie")
*/
public function categorieAction($id)
{
$em = $this->getDoctrine()->getManager();
$repo = $em->getRepository(Produit::class);
$produits = $repo->findByCategorie($id);
return $this->render('front/categorie.html.twig', ['produits' => $produits,'categorie'=>$id]);
}
```

On remarque une nouvelle fonction : ***findByCategorie()***, qui prend en paramètre un objet de la classe **Categorie**. Il faut savoir que cette fonction **n'existe pas** réellement, mais grâce aux méthodes magiques de PHP on peut écrire une fonction ***findByX*** avec ***X*** un attribut de la classe

définie lors de l'instanciation du ***repository*** (dans notre cas c'est Product)

On donne le gabarit de la fiche Catégorie, il faut en extraire les données nécessaires pour créer le vue ***front/categorie.html.twig*** et la modifier pour que les données soient dynamiques :

```
<!DOCTYPE html>
<html>
<head>
<title>GABARIT : fiche Catégorie</title>
<link rel="stylesheet" href="
https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.css">
<meta charset=utf-8>
</head>
<body>
<div class="container">
<div class="jumbotron">
<div class="container">
<h1>Produits de la catégorie Animaux</h1>
<p>Depuis 1998, Animalis propose le meilleur à nos amis les...</p>
<a class="btn btn-info" href="#">Page d'accueil</a>
</div>
</div>
<div class="row">
<div class="col-sm-4">
<div class="thumbnail">
<img src="https://placeimg.com/200/200/animals" alt="">
<div class="caption text-center">
<h3>Chat</h3>
<p>ceci est un chat...ceci est un chat</p>
</div>
</div>
</div>
<div class="col-sm-4">
<img src="https://placeimg.com/200/201/animals" alt="">
<div class="caption text-center">
<h3>Chien</h3>
<p>ceci est unc chien ...ceci est unc chien </p>
</div>
</div>
</div>
</div>
</div>
</body>
</html>
```

Enfin lier le bouton «Produits» de la vue ***front/index.html.twig*** vers la nouvelle vue en passant le bon paramètre. La page qui affiche les produits propose un bouton pour revenir à la page d'accueil.