

Niveau : STS SIO 2^{ème} année☐ Cours ☐ TD ☒ TP

1/26

Module : SLAM5 – Solutions applicatives

Intitulé : Compléments sur Symfony 2

Durée : 2h00 heures

Objectifs :

- ✓ Formulaires CRUD
- ✓ Installation de bundle
- ✓ Programmer au sein d'un Framework



Contenu

Objectif du TP.....	2
Formulaires CRUD.....	2
Création du bundle et des entités et génération des tables	2
Générer un contrôleur CRUD basé sur les entités	5
Allez, à vous	9
Le design.....	18
Installer un bundle	21
Installer Composer.....	21
Installation du bundle « doctrine/doctrine-fixtures-bundle ».....	22
Utilisation de DoctrineFixtureBundle.....	24

Objectif du TP

2/26

Dans ce TP, nous allons créer une petite application permettant de gérer des produits et des catégories de produits.

Enfin, vous n'allez pas faire grand-chose : c'est Symfony qui va faire le travail pour vous. Eh oui, le couple Symfony / Doctrine vous permet de générer, non seulement des tables dans la base de données mais également des formulaires CRUD (Create / Read / Update / Delete) pour vos entités. Autrement dit, ils font tout le travail pour vous.

Dans un second temps, nous verrons comment installer un bundle mis disposition par la communauté très active de Symfony.

N'oubliez pas avec les Framework : lorsque vous voulez faire quelque chose, vérifiez toujours si quelqu'un ne l'a pas déjà fait !

Formulaires CRUD

Création du bundle et des entités et génération des tables

- Créez un répertoire catalogue dans votre dossier www de OneDrive
- Y décompresser l'archive Symfony
- Créez un bundle Sio/CatalogueBundle
- Modifiez le nom de la base de données dans le fichier de paramètre (symfocata)
- Créez une entité Produit

nom	string	50
description	text	
prixHT	decimal	
- Créez une entité Categorie

libelle	string	30
---------	--------	----
- Ouvrez l'entité Produit et positionnez-vous sur le champ prixHT

```
/**
 * @ORM\Column(name="prixHT", type="decimal")
 */
private $prixHT;
```

Si vous laissez ce champ tel quel, vous obtiendrez dans la base de données un champ de table avec le type decimal(10,0) donc sans décimale. Ce qui est un peu gênant pour un prix.

Vous allez donc modifier l'annotation afin de préciser le nombre de décimales souhaitées :

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

- Générez les deux tables
- Ouvrez l'entité produit et ajoutez un attribut \$categorie défini ainsi :

```
/**
 * @var integer
 *
 * @ORM\ManyToOne(targetEntity="Categorie")
 * @ORM\JoinColumn(name="idCategorie", referencedColumnName="id")
 */
private $categorie;
```

- Lancez la génération des getters/setters sur l'entité Produit (examiner les getters/setters générés pour la catégorie).

```
/**
 * Set categorie
 *
 * @param \Sio\CatalogueBundle\Entity\Categorie $categorie
 * @return Produit
 */
public function setCategorie(\Sio\CatalogueBundle\Entity\Categorie $categorie = null)
{
    $this->categorie = $categorie;

    return $this;
}

/**
 * Get categorie
 *
 * @return \Sio\CatalogueBundle\Entity\Categorie
 */
public function getCategorie()
{
    return $this->categorie;
}
```

Vous pouvez constater la catégorie est bien un objet de la classe Categorie !

- Simulez la mise à jour de la BDD

Vous obtenez :

```
C:\...\www\Catalogue>php app/console doctrine:schema:update --dump-sql
ALTER TABLE produit ADD idCategorie INT DEFAULT NULL;
ALTER TABLE produit ADD CONSTRAINT FK_E618D5BBB597FD62 FOREIGN KEY
(idCategorie) REFERENCES Categorie (id);
CREATE INDEX IDX_E618D5BBB597FD62 ON produit (idCategorie);
```

- Mettez à jour la BDD

Vérifier sous PhpMyAdmin :

4/26

#	Nom	Type	Interclassement	Attributs
<input type="checkbox"/> 1	id	int(11)		
<input type="checkbox"/> 2	nom	varchar(50)		
<input type="checkbox"/> 3	description	longtext		
<input type="checkbox"/> 4	prixHT	decimal(10,2)		
<input type="checkbox"/> 5	idCategorie	int(11)		

Relations

Colonne	Contrainte de clé étrangère (INNODB)		
id	symfocata		
nom	Aucun index n'est défini ! Créez-en un dans le dialogue plus bas		
description	Aucun index n'est défini ! Créez-en un dans le dialogue plus bas		
prixHT	Aucun index n'est défini ! Créez-en un dans le dialogue plus bas		
idCategorie	symfocata	categorie	id
Nom de la contrainte FK_E618D5BBB597FD62			
ON DELETE	RESTRICT		
ON UPDATE	RESTRICT		

Générer un contrôleur CRUD basé sur les entités

5/26

- Exécuter la commande : `php app/console doctrine:generate:crud`

```
C:\...\www\Catalogue>php app/console doctrine:generate:crud

Welcome to the Doctrine2 CRUD generator

This command helps you generate CRUD controllers and templates.

First, you need to give the entity for which you want to generate a CRUD.
You can give an entity that does not exist yet and the wizard will help
you defining it.

You must use the shortcut notation like AcmeBlogBundle:Post.

The Entity shortcut name: SioCatalogueBundle:Categorie

By default, the generator creates two actions: list and show.
You can also ask it to generate "write" actions: new, update, and delete.

Do you want to generate the "write" actions [no]? yes

Determine the format to use for the generated CRUD.

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

Determine the routes prefix (all the routes will be "mounted" under this
prefix: /prefix/, /prefix/new, ...).

Routes prefix [/categorie]:

Summary before generation

You are going to generate a CRUD controller for "SioGestionCatalogueBundle:Categorie"
using the "yaml" format.

Do you confirm generation [yes]?

CRUD generation

Generating the CRUD code: OK
Generating the Form code: OK
Confirm automatic update of the Routing [yes]?
Importing the CRUD routes: FAILED

The command was not able to configure everything automatically.
You must do the following changes manually.

- Import the bundle's routing resource in the bundle routing file
(F:\COURS\SIO\sio_2_www\Catalogue\src\Sio\CatalogueBundle/Resources/config/routing.yaml).

SioCatalogueBundle_categorie:
    resource: "@SioCatalogueBundle/Resources/config/routing/categorie.yaml"
    prefix: /categorie

C:\wamp\www\catalogue>
```

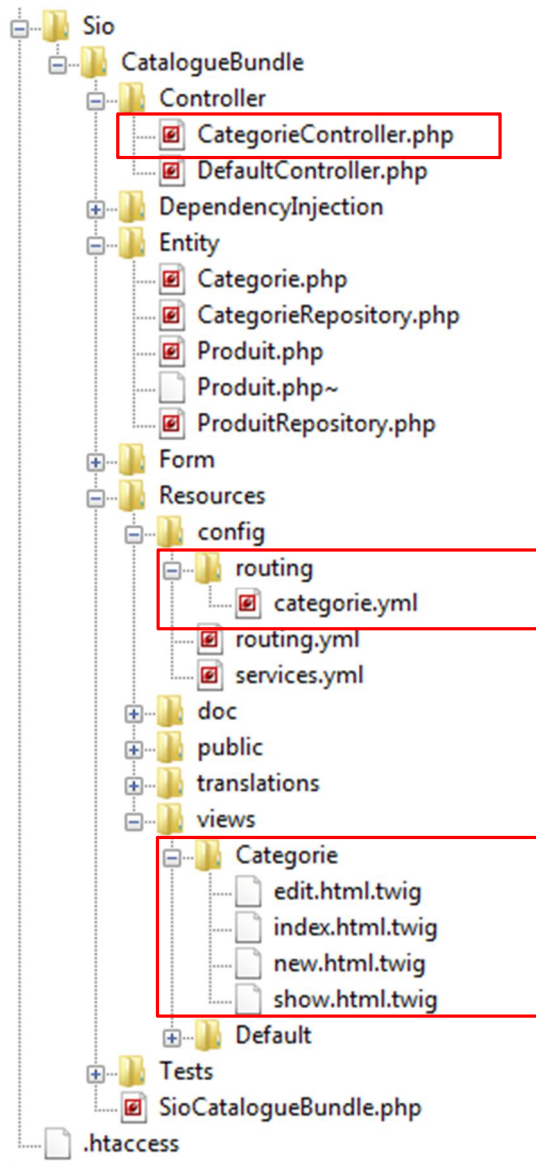
Vous pouvez constater qu'il y a une erreur de routage.

Nous allons corriger cela tout de suite (on aurait pu anticiper et modifier la route avant de lancer cette commande)

- Ouvrez le routeur principal (/app/config/routing.yml) et corrigez-le ainsi :

```
sio_catalogue:  
  resource: "@SioCatalogueBundle/Resources/config/routing.yml"  
  prefix:  /  
  
sio_catalogueBundle_categorie:  
  resource: "@SioCatalogueBundle/Resources/config/categorie/routing.yml"  
  prefix:  /categorie
```

- Examiner le contenu de votre bundle



Le contrôleur a été généré

Les routes ont été générées

Les templates ont été générées

- Examinons les fichiers de routes générés

Le routeur principal /app/config/routing.yml :

```
sio_catalogueBundle_categorie:  
  resource: "@SioCatalogueBundle/Resources/config/routing/categorie.yml"  
  prefix:   /categorie
```

Il nous indique que toutes les routes du fichier categorie.yml doivent être préfixées par « /categorie ».

Le fichier de routes de notre contrôleur categorie.yml :

```
categorie:  
  path:      /  
  defaults: { _controller: "SioCatalogueBundle:Categorie:index" }  
  
categorie_show:  
  path:      /{id}/show  
  defaults: { _controller: "SioCatalogueBundle:Categorie:show" }  
  
categorie_new:  
  path:      /new  
  defaults: { _controller: "SioCatalogueBundle:Categorie:new" }  
  
categorie_create:  
  path:      /create  
  defaults: { _controller: "SioCatalogueBundle:Categorie:create" }  
  requirements: { _method: post }  
  
categorie_edit:  
  path:      /{id}/edit  
  defaults: { _controller: "SioCatalogueBundle:Categorie:edit" }  
  
categorie_update:  
  path:      /{id}/update  
  defaults: { _controller: "SioCatalogueBundle:Categorie:update" }  
  requirements: { _method: post|put }  
  
categorie_delete:  
  path:      /{id}/delete  
  defaults: { _controller: "SioCatalogueBundle:Categorie:delete" }  
  requirements: { _method: post|delete }
```

Devant chaque route (path) du fichier, il faut donc ajouter /categorie.

Voyons un peu la première route :

```
categorie:  
  path:      /  
  defaults: { _controller: "SioCatalogueBundle:Categorie:index" }
```


Le chemin est « / ». Pour appeler la page correspondant à l'action « index », il faut donc entrer l'url suivante :

```
http://localhost/votreAlias/Catalogue/web/app_dev.php/categorie/
```

De la même manière, pour appeler la page correspondant à l'action « show », il faut entrer l'url suivante :

```
http://localhost/votreAlias/Catalogue/web/app_dev.php/categorie/{id}/show
```

De la même manière, pour appeler la page correspondant à l'action « new », il faut entrer l'url suivante :

```
http://localhost/votreAlias/Catalogue/web/app_dev.php/categorie/new
```

- On teste ?

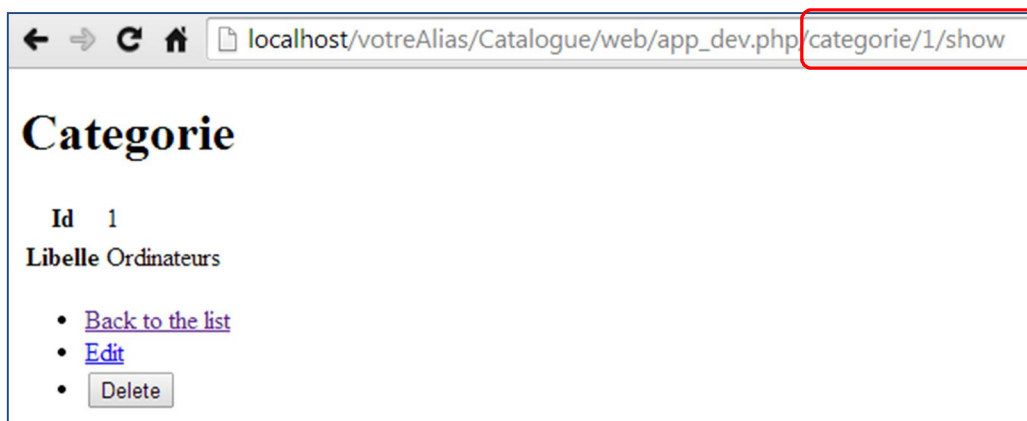
Allez, on ouvre la page principale :

```
http://localhost/votreAlias/Catalogue/web/app_dev.php/categorie/new
```



Saisissez la catégorie « Ordinateurs ».

Vous êtes redirigé vers la page détail de la catégorie que vous venez de créer :



Saisissez maintenant les catégories « Logiciels » et « Imprimantes ».

Retour sur la page de démarrage. Vous obtenez :

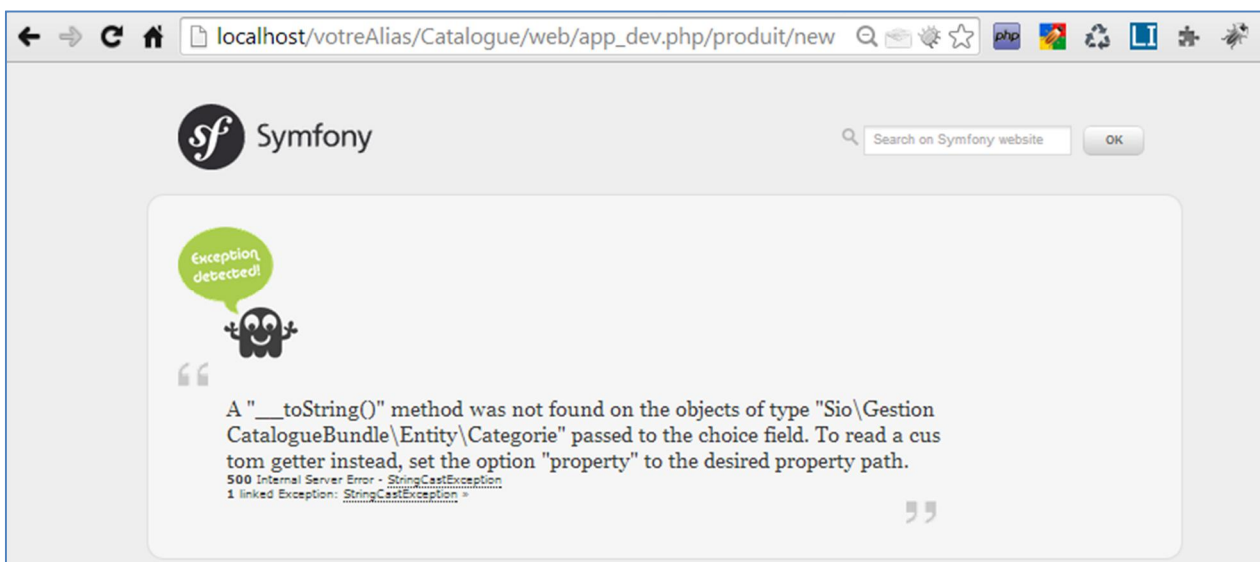


Tout est opérationnel, y compris la persistance des données et la navigation !

Fabuleux, n'est-ce pas ?

Allez, à vous ...

- Générez le contrôleur CRUD pour l'entité produit. N'oubliez pas de modifier le routeur principal
- Ouvrez la page permettant de créer un nouveau produit. Vous obtenez une erreur :



Voyons pourquoi.

Mais avant cela il faut d'abord regarder comment le code est organisé.

Rappelez-vous dans le TP précédent ! Notre action « ajouter » était codée ainsi :

```

public function ajouterAction()
{
    // On crée un objet Abonne
    $abonne = new Abonne();

    // On crée un FormBuilder et on lui ajoute les
    // champs que l'on souhaite
    $formBuilder = $this->createFormBuilder($abonne)
        ->add('nom', 'text', array('label' => 'Nom'))
        ->add('prenom', 'text', array(
            'label' => 'Prénom',
            'required' => false))
        ->add('dateNaissance', 'birthday', array(
            'label' => 'Date de Naissance',
            'widget' => 'single_text'))
        ->add('email', 'email', array('label' => 'Adresse mail'))
        ->add('Enregistrer', 'submit');

    // On génère le formulaire à partir du formBuilder
    $form = $formBuilder->getForm();

    // On récupère la requête
    $request = $this->get('request');

    // On vérifie qu'elle est de type POST
    if ($request->getMethod() == 'POST') {

        // On fait le lien Requête <-> Formulaire
        // À partir de maintenant, la variable $abonne contient les
        // valeurs entrées dans le formulaire par le visiteur
        $form->bind($request);

        // On vérifie que les valeurs entrées sont correctes
        if ($form->isValid()) {
            $em = $this->getDoctrine()->getManager();
            $em->persist($abonne);
            $em->flush();

            // On définit un message flash
            $this->get('session')->getFlashBag()->add('info', 'Abonné bien
ajouté');

            // On redirige vers la page de visualisation de l'abonne'
            // nouvellement créé
            return $this->redirect( $this->generateUrl('sioinscriptions_voir',
                array('id' => $abonne->getId())) );
        }
    }

    // On passe la méthode createView() du formulaire à la vue afin qu'elle
    // puisse l'afficher
    return $this->render('SioInscriptionsBundle:Inscriptions:ajouter.html.twig',
        array('form' => $form->createView()));
}

```

Celle-du TP actuel ressemble à ceci :

```
public function newAction()
{
    $entity = new Produit();
    $form = $this->createCreateForm($entity);

    return $this->render('SioCatalogueBundle:Produit:new.html.twig', array(
        'entity' => $entity,
        'form' => $form->createView(),
    ));
}
```

Très réduite, n'est-ce pas ?

Tout d'abord, la création du formulaire est déléguée à une autre fonction : `createCreateForm()`

```
private function createCreateForm(Produit $entity)
{
    $form = $this->createForm(new ProduitType(), $entity, array(
        'action' => $this->generateUrl('produit_create'),
        'method' => 'POST',
    ));

    $form->add('submit', 'submit', array('label' => 'Create'));

    return $form;
}
```

Nous allons revenir dessus.

Mais avant, pourquoi ne trouve-t-on pas le traitement de la réponse dans cette méthode ?

Si vous examinez la méthode `createCreateForm`, vous pouvez remarquer que l'on passe au formulaire une action à laquelle on associe l'url correspondant à la route `produit_create`

Regardez dans le fichier de routes `produit.yml` :

```
produit_create:
    path: /create
    defaults: { _controller: "SioCatalogueBundle:Produit:create" }
    requirements: { _method: post }
```

Cette route correspond à l'action `create` de notre contrôleur :

```

public function createAction(Request $request)
{
    $entity = new Produit();
    $form = $this->createCreateForm($entity);
    $form->handleRequest($request);

    if ($form->isValid()) {
        $em = $this->getDoctrine()->getManager();
        $em->persist($entity);
        $em->flush();

        return $this->redirect($this->generateUrl('produit_show',
            array('id' => $entity->getId())));
    }

    return $this->render('SioCatalogueBundle:Produit:new.html.twig',
        array(
            'entity' => $entity,
            'form'   => $form->createView(),
        ));
}

```

Et c'est donc cette action qui gère la réponse à la requête.

C'est une autre façon de faire les choses, qui présente comme avantage d'être plus modulaire et donc de permettre de réutiliser des fonctionnalités en différents endroits de l'application. On pourrait par exemple imaginer une autre page, ailleurs, qui permettrait également de créer un produit. Pour traiter la réponse à l'action correspondante, il suffirait donc d'appeler la méthode `createAction` du contrôleur.

Voilà, c'était un petit aparté sur la structure du code et pour vous expliquer la présence d'autant de routes dans le fichier de routes.

Une dernière remarque sur cette route particulière :

```

produit_create:
    path:      /create
    defaults: { _controller: "SioCatalogueBundle:Produit:create" }
    requirements: { _method: post }

```

Notez la présence de l'option « requirements » : elle permet de préciser des contrôles supplémentaires à effectuer sur les routes.

On pourrait par exemple vérifier qu'un paramètre {id} est numérique, qu'il est numérique, etc.

Ici, on précise que la méthode pour cette route est obligatoirement « post ».

Ainsi, si vous entrez l'url http://localhost/votreAlias/Catalogue/web/app_dev.php/produit/create, vous obtenez l'erreur suivante (puisque, via l'url, une requête est transmise obligatoirement par la méthode get) :



Revenons à la création de notre formulaire :

```
private function createCreateForm(Produit $entity)
{
    $form = $this->createForm(new ProduitType(), $entity, array(
        'action' => $this->generateUrl('produit_create'),
        'method' => 'POST',
    ));

    $form->add('submit', 'submit', array('label' => 'Create'));

    return $form;
}
```

Dans le TP précédent, nous avons :

```
public function ajouterAction()
{
    $abonne = new Abonne();

    $formBuilder = $this->createFormBuilder($abonne)
        ->add('nom', 'text', array('label' => 'Nom'))
        ->add('prenom', 'text', array(
            'label' => 'Prénom',
            'required' => false))
        ->add('dateNaissance', 'birthday', array(
            'label' => 'Date de Naissance',
            'widget' => 'single_text'))
        ->add('email', 'email', array('label' => 'Adresse mail'))
        ->add('Enregistrer', 'submit');

    $form = $formBuilder->getForm();
}
```

En fait nous n'avons pas utilisé la même méthode.

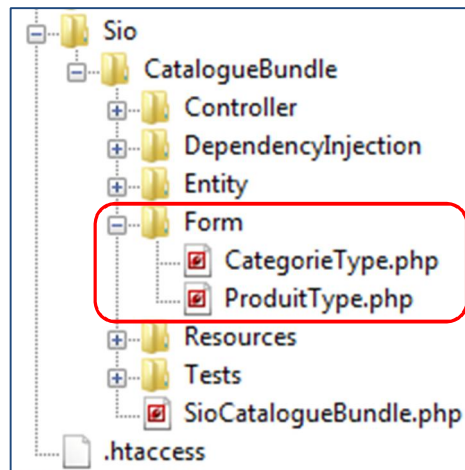
Le générateur de contrôleur utilise la méthode `createForm()` du gestionnaire d'entité.

Cette méthode permet de créer des formulaires réutilisables.

Pour cela, il faut détacher la définition du formulaire dans une classe à part, nommée, par convention, *nomEntiteType*, ici donc il s'agit de la classe `ProduitType` :

```
private function createcreateForm(Produit $entity)
{
    $form = $this->createForm(new ProduitType(), $entity, array(
        'action' => $this->generateUrl('produit_create'),
        'method' => 'POST',
    ));
}
```

Le fichier contenant cette classe se situe dans le dossier Form du bundle :



Ouvrez ce fichier (ProduitType.php).

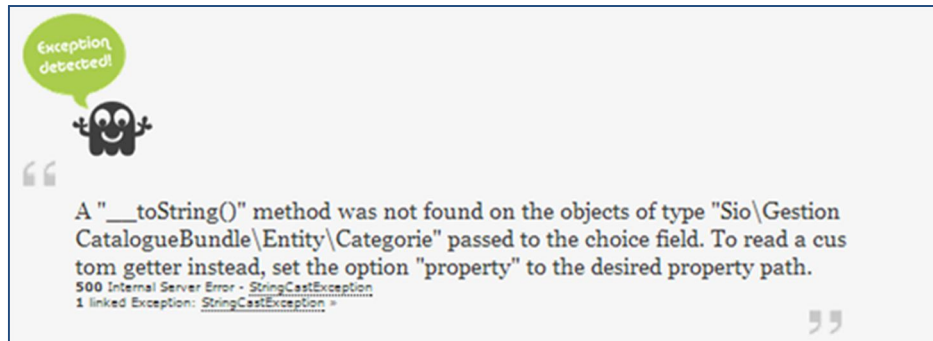
La méthode qui nous intéresse est la méthode `buildForm`. C'est elle qui construit le formulaire en lui passant les différents champs.

Mais comme vous pouvez le constater, le type des champs n'est pas précisé. Donc, par défaut, le composant Form va les deviner à partir des annotations Doctrine qu'on a mises dans l'objet.

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

Or, ici, c'est la catégorie qui pose problème. En effet, il s'agit d'un objet. Le composant Form ne sait pas comment transformer ce champ et, par défaut, tente de le transformer en champ de type text.

D'où l'erreur obtenue :



Nous allons dans un premier temps suivre les recommandations de Symfony et créer une méthode `__toString()` dans notre entité `Categorie`.

Ouvrez le fichier `/Sio/CatalogueBundle/Entity/Categorie.php` et ajoutez la méthode suivante :

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

Testez la page permettant de créer un nouveau produit. Vous obtenez maintenant :

Et oui, une liste déroulante !

La seconde méthode consiste à définir correctement les types dans le constructeur du formulaire (c'est mieux !).

Tout d'abord, supprimez la méthode `__toString()` que nous venons d'ajouter à la classe `Categorie` (histoire de s'assurer que nos nouvelles modifications fonctionnent).

Ouvrez maintenant le fichier `/Sio/CatalogueBundle/Form/ProduitType.php`. Nous allons en profiter pour définir le type de tous les champs.

16/26

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('nom', 'text', array('label' => 'Nom'))
        ->add('description', 'textarea', array('label' => 'Description'))
        ->add('prixHT', 'money', array(
            'label' => 'Prix hors taxe ',
            'currency' => 'EUR',
            'precision' => 2))
        ->add('categorie', 'entity', array(
            'label' => 'Catégorie',
            'class' => 'SioCatalogueBundle:Categorie',
            'property' => 'libelle',
        ));
}
```

Testez la page.

Vous obtenez le même résultat qu'avec la méthode `__toString()` sauf que, en plus d'avoir des labels un peu plus propres, nous avons été plus précis au niveau du champ `prixHT`.

Ainsi, Symfony arrondira la valeur saisie à 2 décimales, sans aucune ligne de code supplémentaire.

Vous pouvez également remarquer la présence du sigle € (lié au type de champ `money`).

Produit creation

Nom

Description

Prix hors taxe €

Catégorie

- [Back](#)

Produit edit

Nom

Description

Prix hors taxe €

Catégorie

- [Back to the list](#)
-

En procédant ainsi, nous pouvons également agir sur les labels.

Autre avantage : les modifications effectuées ici pour le formulaire permettant de saisir un produit seront également valable pour le formulaire de modification !

17/26

De la même façon, vous allez modifier le constructeur de formulaire de l'entité Catégorie :

```
public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('libelle', 'text', array('label' => 'Libellé'));
}
```

Pour terminer, il reste à gérer le contrôleur par défaut : pour l'instant, il concerne les url's de type /hello/{name}.

- Vérifiez le routeur principal : /app/config/routing.yml

```
sio_catalogue:
    resource: "@SioCatalogueBundle/Resources/config/routing.yml"
    prefix: /

sio_catalogueBundle_categorie:
    resource: "@SioCatalogueBundle/Resources/config/routing/categorie.yml"
    prefix: /categorie

sio_catalogueBundle_produit:
    resource: "@SioCatalogueBundle/Resources/config/routing/produit.yml"
    prefix: /produit
```

- Puis modifiez le fichier de routes par défaut :
/Sio/CatalogueBundle/Resources/config/routing.yml

```
Sio_accueil:
    path: /
    defaults: { _controller: SioCatalogueBundle:Default:index }
```

- Enfin, modifiez le contrôleur par défaut pour supprimer le paramètre *name* :
\Sio\CatalogueBundle\Controller\DefaultController.php

```
class DefaultController extends Controller
{
    public function indexAction()
    {
        return $this->render('SioCatalogueBundle:Default:index.html.twig');
    }
}
```

Voilà. Vous avez construit plusieurs pages d'un site web dynamique en quelques commandes console et très peu de code.

Le design

18/26

Bon évidemment, il reste la partie design, mais comme nous l'avons vu dans le TP précédent, ça peut aller très vite.

Ce n'est pas l'objet de ce TP. Je vais donc vous fournir les fichiers.

Décompressez l'archive bootstrap dans le répertoire web de votre application.

Récupérez le dossier design.

Dans ce dossier, vous trouverez des répertoires contenant les fichiers à récupérer (nouveaux ou modifiés).

Le nom des répertoires vous indique l'emplacement cible des fichiers.

Attention de ne pas vous tromper.

Un petit test pour vérifier :

- La page d'accueil (contrôleur Default)



- La gestion des produits (contrôleur Produit)

localhost/votreAlias/Catalogue/web/app_dev.php/produit/

Gestion catalogue SIO

Liste des produits

Id	Nom	Description	Prixht	Actions
5	MS Word	Microsoft Word 2013	79.45	<ul style="list-style-type: none">voirmodifier

[Créer un nouveau produit](#)

localhost/votreAlias/Catalogue/web/app_dev.php/produit/5/show

Gestion catalogue SIO

Détail produit

Id	5
Nom	MS Word
Description	Microsoft Word 2013
Prixht	79.45

[Retour à la liste](#)
[Modifier](#)
[Supprimer](#)

localhost/votreAlias/Catalogue/web/app_dev.php/produit/new

Gestion catalogue SIO

Création de produit

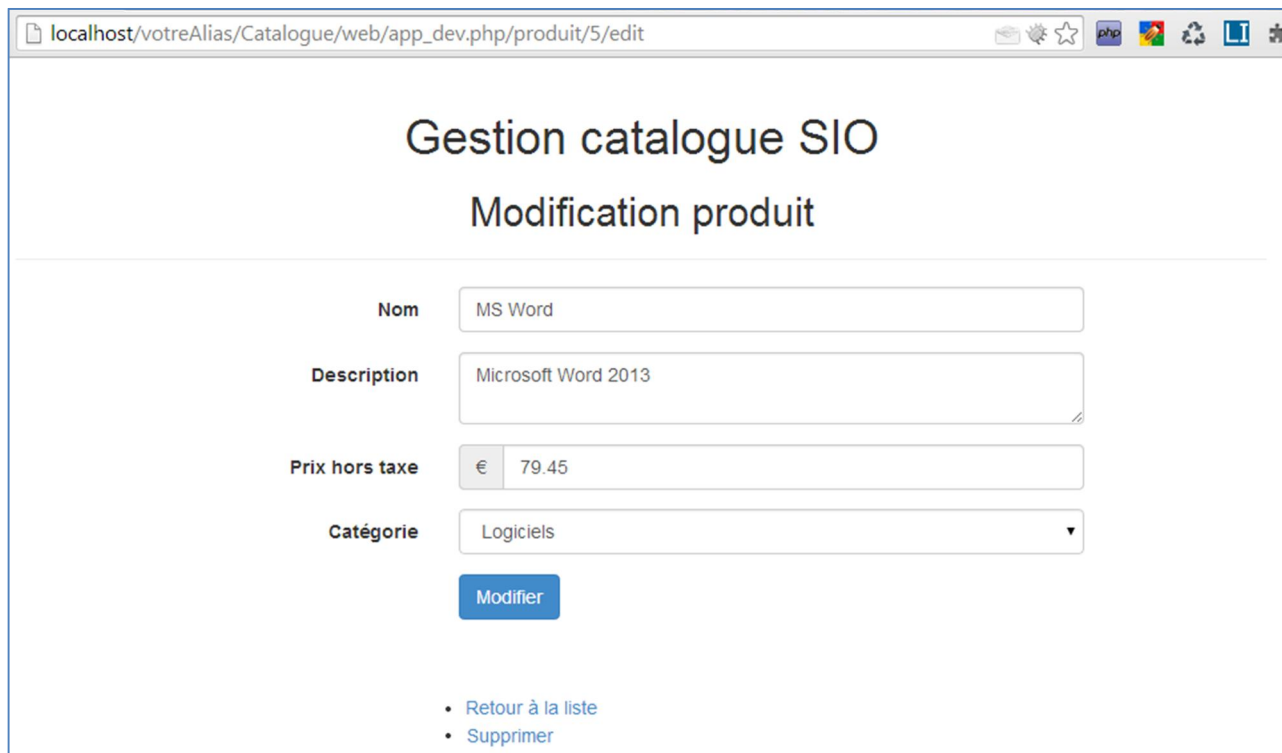
Nom

Description

Prix hors taxe €

Catégorie

[Retour à la liste](#)



localhost/votreAlias/Catalogue/web/app_dev.php/produit/5/edit

Gestion catalogue SIO

Modification produit

Nom	<input type="text" value="MS Word"/>
Description	<input type="text" value="Microsoft Word 2013"/>
Prix hors taxe	€ <input type="text" value="79.45"/>
Catégorie	<input type="text" value="Logiciels"/>

- [Retour à la liste](#)
- [Supprimer](#)

- Et idem pour la gestion des catégories (contrôleur `Categorie`)

Un peu de ménage ...

Avant de poursuivre, vous allez supprimer le bundle de démonstration `Acme`.

N'oubliez pas de modifier le `Kernel` et le fichier de `Routing` spécifique à l'environnement de développement.

Installer un bundle

Nous allons voir maintenant comment installer un bundle

Pour ce faire nous allons utiliser Composer, c'est la méthode la plus simple.

Composer est un outil pour gérer les dépendances en PHP. Les dépendances, dans un projet, ce sont toutes les bibliothèques dont votre projet dépend pour fonctionner. Par exemple, votre projet utilise la bibliothèque `SwiftMailer` pour envoyer des e-mails, il « dépend » donc de `SwiftMailer`.

Composer a pour objectif de vous aider à gérer toutes vos dépendances. En effet, il y a plusieurs problématiques lorsqu'on utilise des bibliothèques externes :

- Ces bibliothèques sont mises à jour. Il vous faut donc les mettre à jour une à une pour vous assurer de corriger les bogues de chacune d'entre elles.
- Ces bibliothèques peuvent elles-mêmes dépendre d'autres bibliothèques. Dans ce cas, il faut gérer l'ensemble de ces dépendances (installation, mises à jour, etc.).

Installer Composer

- Tout d'abord, il faut installer Git sur votre machine (au lycée, c'est fait !). Git est un gestionnaire de versions et Composer l'utilise pour télécharger les bibliothèques.

Sous Windows, il faut utiliser `msysgit`. Cela installe un système d'émulation des commandes Unix sous Windows et Git lui-même.

Pour le télécharger, c'est là : <http://msysgit.github.io/>

- Une fois cela fait, il faut ajouter le répertoire contenant les exécutables Git au PATH de Windows. Ajoutez donc ceci : « `;C:\Program Files (x86)\Git\bin` » à la suite de votre variable d'environnement utilisateur « PATH ». Puis fermez la boîte de dialogue pour que la modification soit prise en compte.

Si vous ajoutez le chemin dans la variable d'environnement système et non dans la variable d'environnement utilisateur, vous devrez ensuite, redémarrer votre machine.

Exécutez la commande « `git version` » dans la console pour vérifier que tout fonctionne correctement.

- Ensuite, il faut installer Composer.

Téléchargez le fichier `composer.phar` à l'adresse <https://packagist.org/> (au lycée, récupérez ce fichier sur agora).

Il suffit de placer le fichier `composer.phar` à la racine de votre site :
`c:\...\OneDrive\www\catalogue\`

Installation du bundle « doctrine/doctrine-fixtures-bundle »

Nous allons maintenant procéder à l'installation du bundle DoctrineFixtureBundle. Ce bundle permet de générer des scripts pour insérer des données dans la base. L'intérêt est de pouvoir recharger les données chaque fois que nécessaire (les anciennes sont supprimées automatiquement). Un autre intérêt est que l'on manipule des objets : plus besoin de requêtes SQL.

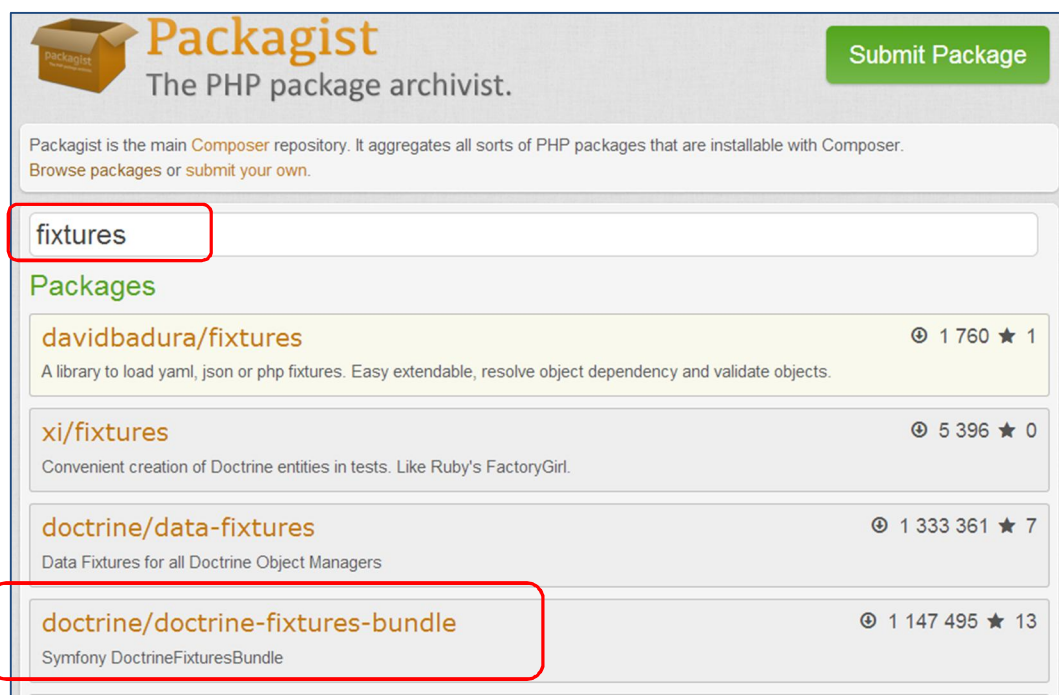
➤ Trouver le nom exact du bundle et sa version

Pour installer un Bundle, il est nécessaire de connaître son nom exact.

Le site <https://packagist.org/> référence des bundles pour Symfony ou d'autres Frameworks et vous permet de retrouver leur nom exact ainsi que la version qui vous intéresse (qui dépendra bien évidemment de la version de Symfony et de doctrine que vous utilisez).

Pour trouver des bundles Symfony, vous avez également à votre disposition le site <http://knpbundles.com/>.

Rendez-vous sur le site [Packagist](https://packagist.org/) et tapez « fixture » dans la zone de recherche sans faire « Enter ».



Vous trouvez dans la liste le bundle qui nous intéresse. Si vous cliquez dessus, vous arrivez sur une page qui vous présente toutes les versions disponibles du bundle, les versions minimums des différents outils utilisés (Symfony, doctrine, ...).

Pour notre part, nous allons utiliser la version dev-master (version en cours de développement) mais qui nous convient tout à fait.

➤ Déclarer les dépendances

Cela se fait dans le fichier `composer.json`, qui se trouve à la racine de votre site et qui contient les informations sur les bibliothèques dont dépend votre projet ainsi que leur version.

Ouvrez, à l'aide d'un éditeur, le fichier `composer.json` qui se trouve à la racine de votre application. Puis, dans la section « `require` », ajoutez une référence à un bundle, comme ci-après :

```
"require": {
    "php": ">=5.3.3",
    "symfony/symfony": "2.5.*",
    "doctrine/orm": "~2.2,>=2.2.3",
    "doctrine/doctrine-bundle": "~1.2",
    "twig/extensions": "~1.0",
    "symfony/assetic-bundle": "~2.3",
    "symfony/swiftmailer-bundle": "~2.3",
    "symfony/monolog-bundle": "~2.4",
    "sensio/distribution-bundle": "~3.0",
    "sensio/framework-extra-bundle": "~3.0",
    "incenteev/composer-parameter-handler": "~2.0",
    "doctrine/doctrine-fixtures-bundle": "dev-master"
},
```

N'oubliez la virgule
à la fin de la ligne
précédente

➤ Mettre à jour les dépendances

Ouvrez une console, placez-vous à la racine de votre site puis exécutez :

```
php composer.phar update doctrine/doctrine-fixtures-bundle
```

Si tout s'est bien passé (aucune erreur), le dossier `doctrine-fixtures-bundle` a dû être créé dans le dossier `/vendor/doctrine`.

Remarque :

Si vous voulez mettre à jour toutes vos dépendances, exécutez :

```
php composer.phar update
```

Si vous voulez mettre à jour Composer lui-même, exécutez :

```
php composer.phar update --self
```

➤ Enregistrer le bundle dans le Kernel

Ouvrez le fichier `/app/AppKernel.php` et déclarez le bundle comme ci-dessous :

```
public function registerBundles()
{
    $bundles = array(
        new Symfony\Bundle\FrameworkBundle\FrameworkBundle(),
        new Symfony\Bundle\SecurityBundle\SecurityBundle(),
        new Symfony\Bundle\TwigBundle\TwigBundle(),
        new Symfony\Bundle\MonologBundle\MonologBundle(),
        new Symfony\Bundle\SwiftmailerBundle\SwiftmailerBundle(),
        new Symfony\Bundle\AsseticBundle\AsseticBundle(),
        new Doctrine\Bundle\DoctrineBundle\DoctrineBundle(),
        new Doctrine\Bundle\FixturesBundle\DoctrineFixturesBundle(),
        new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
        new Sio\GestionCatalogueBundle\SioGestionCatalogueBundle(),
        new Sio\UserBundle\SioUserBundle(),
    );

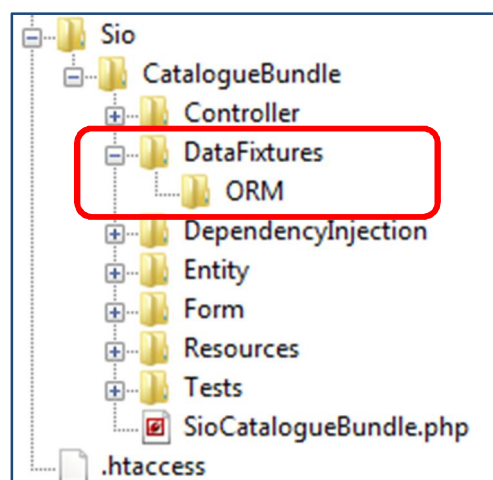
    if (in_array($this->getEnvironment(), array('dev', 'test'))) {
        $bundles[] = new Symfony\Bundle\WebProfilerBundle\WebProfilerBundle();
        $bundles[] = new Sensio\Bundle\DistributionBundle\SensioDistributionBundle();
        $bundles[] = new Sensio\Bundle\GeneratorBundle\SensioGeneratorBundle();
    }
}
```

Et voilà, notre bundle est opérationnel.

Utilisation de DoctrineFixtureBundle

Maintenant que tout est en place, nous allons pouvoir créer notre fichier de fixtures. Pour que Symfony puisse les trouver, il faut créer un dossier particulier dans notre Bundle.

- Placez-vous dans le répertoire de votre bundle `GestionCatalogueBundle` et créez le dossier `DataFixtures`. A l'intérieur de ce dossier créez un nouveau dossier `ORM`. Attention, cette structure est obligatoire.



Dans ce dossier `ORM`, vous pouvez créer des fichiers PHP de n'importe quel nom, ils seront reconnus par Symfony comme fichier de fixtures.

- Sous ORM, créez un nouveau fichier LoadCategoriesProduits.php. Voici le contenu de ce fichier :

```
<?php
namespace Sio\UserBundle\DataFixtures\ORM;

use Doctrine\Common\DataFixtures\FixtureInterface;
use Doctrine\Common\Persistence\ObjectManager;
use Sio\CatalogueBundle\Entity\Categorie;
use Sio\CatalogueBundle\Entity\Produit;

class LoadCategoriesProduits implements FixtureInterface
{
    // Dans l'argument de la méthode load, l'objet $manager est
    // l'EntityManager
    public function load(ObjectManager $manager)
    {
        /**
         * Création des catégories
         */
        $ordi = new Categorie();
        $ordi->setLibelle("Ordinateurs");
        $manager->persist($ordi);
        $manager->flush();

        $impr = new Categorie();
        $impr->setLibelle("Imprimantes");
        $manager->persist($impr);
        $manager->flush();

        $log = new Categorie();
        $log->setLibelle("Logiciels");
        $manager->persist($log);
        $manager->flush();

        /**
         * Création des produits
         */
        $p1 = new Produit();
        $p1->setNom("Word");
        $p1->setDescription("Microsoft Word 2013 - Traitement de texte");
        $p1->setPrixHT(79.99);
        $p1->setCategorie($log);
        $manager->persist($p1);

        $p2 = new Produit();
        $p2->setNom("HP 4200");
        $p2->setDescription("HP Deskjet 3 en 1 F4200");
        $p2->setPrixHT(87);
        $p2->setCategorie($impr);
        $manager->persist($p2);

        $p3 = new Produit();
        $p3->setNom("HP Pavilion DV7");
        $p3->setDescription("PC portable core i7 - DD 1T - RAM 4G ");
        $p3->setPrixHT(87);
        $p3->setCategorie($ordi);
        $manager->persist($p3);

        $manager->flush();
    }
}
```

- Ouvrez la console, placez-vous à la racine de votre site puis exécutez la commande suivante, qui permet de lancer les fixtures

```
php app/console doctrine:fixtures:load
```

Le contenu de votre base de données a été entièrement remplacé. Vous pouvez le vérifier sous PhpMyAdmin.

L'intérêt ici ne vous saute peut-être pas aux yeux, mais c'est très pratique :

- Lorsque vous testez votre application, vous êtes amenés à créer, modifier et supprimer des données. Et il est souvent nécessaire de repartir sur une base propre. Avec les fixtures, une commande à lancer et c'est chose faite.
- Lorsque vous déplacez votre application sur une autre machine (chez vous par exemple, parce que vous voulez vous avancer sur votre TP), vous avez juste à copier le dossier de votre application et à lancer trois petites commandes :

```
app/console doctrine:database:create
```

➔ pour créer la base de données

```
app/console doctrine:schema:update --force
```

➔ pour générer les tables

```
php app/console doctrine:fixtures:load
```

➔ pour insérez les données

C'est bon, vous me suivez toujours ? Vous êtes prêt pour la dernière ligne droite ?

Alors, suite au prochain épisode.

