M4D203D - POO Avancée

David Annebicque

2016

1 Sujet pour les TP

Proposer une application qui permette de gérer des programmes qui sont dans des catégories. Ces programmes peuvent être placés dans une grille.

L'application devra permettre :

- Ajouter, modifier, supprimer une catégorie. La suppression d'une catégorie ne sera possible que si aucun programme n'est associé à cette catégorie.
- Ajouter, modifier, supprimer un programme. La suppression d'un programme impliquera sa suppression de la grille.
- Ajouter, modifier, supprimer des créneaux dans la grille.
- Affecter, supprimer un programme de la grille.
- Afficher la grille complète.
- Afficher un programme précis.
- Afficher une catégorie avec les programmes associés.
- L'ensemble de cette partie sera protégée par un login et un mot de passe (séance 10).

2 Découverte du CRUD

2.1 Utiliser la console

CRUD : "Create Read Update Delete".

Symfony permet de générer automatiquement l'ensemble des fichiers nécessaires pour créer un formulaire d'ajout, une liste des éléments de la table, un formulaire de modification et un bouton pour supprimer ou voir un élément spécifique.

Cette fonctionnalité est pratique pour des cas classiques. Cependant il est important de comprendre la logique du code produit afin de pouvoir le modifier et l'adapter aux besoins spécifiques de votre application.

Dans la console écrire la ligne suivante, en rapport avec une entité précise :

```
1 php bin/console doctrine:generate:crud AppBundle:Categories
```

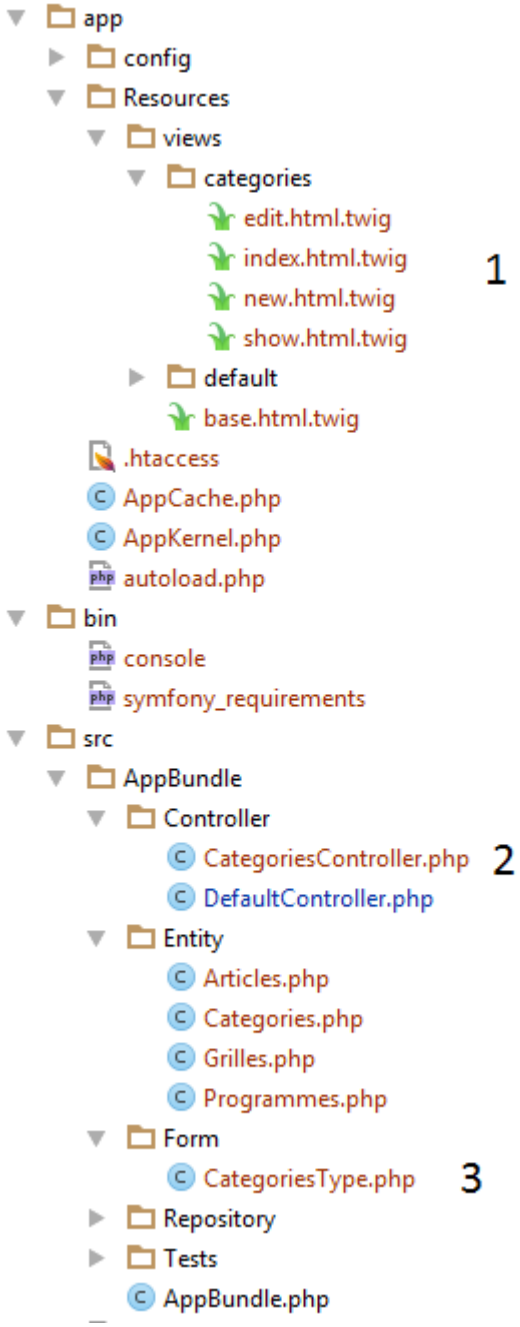
Vous devez ensuite répondre à plusieurs questions dans la console. Les questions sont les suivantes :

- **The Entity shortcut name**: Comme il est précisé dans l'appel de la commande, pas nécessaire de remplir ce champs. Néanmoins, en cas d'erreur vous pouvez indiquer une autre entité ici.
- **Do you want to generate the "write" actions [no]?** : Par défaut, le CRUD ne génére que les actions de lire. Si vous voulez aussi un create et un update vous devez répondre "yes".
- **Configuration format (yml, xml, php, or annotation) [annotation]** : On garde le choix par défaut, pour avoir les routes dans les commentaires (annotations).
- **Routes prefix [/categories]:** : Permet de définir à quoi va ressembler l'URL. Vous pouvez modifier librement, sachant qu'il sera possible de modifier ultérieurement, facilement.

Vous devriez obtenir un affichage similaire au résultat ci-dessous à la fin de la génération du CRUD.

```
C:\wamp\www\s4D>php bin/console doctrine:generate:crud AppBundle:Categories


  Welcome to the Doctrine2 CRUD generator



This command helps you generate CRUD controllers and templates.

First, give the name of the existing entity for which you want to generate a CRUD
(use the shortcut notation like AcmeBlogBundle:Post)

The Entity shortcut name [AppBundle:Categories]:

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 (yml, xml, php, or annotation) [annotation]:

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

Routes prefix [/categories]:


  Summary before generation


You are going to generate a CRUD controller for "AppBundle:Categories"
using the "annotation" format.

Do you confirm generation [yes]?


  CRUD generation


Generating the CRUD code: OK
Generating the Form code: OK
Updating the routing: OK


  Everything is OK! Now get to work :).
```

A ce stade, Symfony a généré de nombreux fichiers :



1. Les vues :
   — edit : Le formulaire de modification
   — index : La page d'accueil
   — new : Le formulaire pour la création d'un nouvel item
   — show : La page qui détail un item en particulier.

2. Le contrôleur qui va contenir les différentes méthodes et les routes.
3. Le formType qui décrit le formulaire à afficher en création et en update (même formulaire)

3 Comprendre le code généré

Il est intéressant d'étudier et de comprendre le code générer. Cette partie s'intéresse donc au code généré dans le contrôleur, le formType et les vues. La compréhension du code est essentiel afin de pouvoir modifier et adapter le fonctionnement généré automatiquement à votre application.

3.1 Le contrôleur

3.1.1 L'en-tête

```
1 <?php
2 use Symfony\Component\HttpFoundation\Request;
3 use Symfony\Bundle\FrameworkBundle\Controller\Controller;
4 use
  ↪ Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
5 use
  ↪ Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
6 use AppBundle\Entity\Categories;
7 use AppBundle\Form\CategoriesType;
```

Dans cette première partie du code du contrôleur, il faut définir les éléments nécessaires au bon fonctionnement :

— ligne 2 : Le composant Request permet de récupérer les éléments envoyés par le navigateur du client, et notamment les données issues d'un formulaire.
— ligne 3 : Ligne toujours présente dans un contrôleur, permet d'utiliser les méthodes de Symfony.
— ligne 4 : Cette ligne permet d'utiliser l'annotation @Method qui permet de spécifier comme la route doit être appelée. Soit en GET (URL directe), soit en POST (par l'intermédiaire d'un formulaire), ... Cela fait référence aux API REST.
— ligne 5 : Permet d'utiliser les routes dans les annotations.
— ligne 6 : Précise à Symfony et au contrôleur que l'on va utiliser l'entité Categories, ce qui va permettre d'alléger l'écriture par la suite.

— ligne 7 : Précise à Symfony et au contrôleur que l'on va utiliser le formType (déclaration de formulaire) CategoriesType, ce qui va permettre d'alléger l'écriture par la suite.

```
1  <?php
2  /**
3  * Categories controller.
4  *
5  * @Route("/categories")
6  */
7  class CategoriesController extends Controller
```

Dans cette en-tête de la classe, l'annotation @Route précise le préfixe de l'URL qui est appliqué à l'ensemble des méthodes de cette classe. Autrement-dit, toutes les méthodes vont commencées par domaine.tld/categories/... Si vous souhaitez modifier ce préfixe c'est ici qu'il faut le changer.

3.1.2 La méthode index

Cette méthode permet l'affichage d'un tableau avec l'ensemble des enregistrements de la table concernée.

```
1  <?php
2  /**
3  * Lists all Categories entities.
4  *
5  * @Route("/", name="categories_index")
6  * @Method("GET")
7  */
8  public function indexAction()
9  {
10  $em = $this->getDoctrine()->getManager();
11
12  $categories =
    ↪  $em->getRepository('AppBundle:Categories')->findAll();
13
14  return $this->render('categories/index.html.twig', array(
15    'categories' => $categories,
16    ));
17 }
```

— Ligne 6 : permet de spécifier de quelle manière peut être appelée la

route. Ici uniquement en méthode GET, c'est à dire par l'URL directement.
— Ligne 10 : récupération du manager pour manipuler la base de données.
— Ligne 12 : requête qui permet de sélectionner tous les éléments de la table Catégories.
— Ligne 14 : affiche la vue, en passant un tableau contenant tous les enregistrements de la table.

3.1.3 La méthode new

Cette méthode permet de créer un formulaire, de gérer l'enregistrement d'un nouvel objet, et d'afficher la page pour la création de ce nouvel objet.

```
1  <?php
2  /**
3  * Creates a new Categories entity.
4  *
5  * @Route("/new", name="categories_new")
6  * @Method({"GET", "POST"})
7  */
8  public function newAction(Request $request)
9  {
10   $category = new Categories();
11   $form =
     ↪  $this->createForm('AppBundle\Form\CategoriesType',
     ↪  $category);
12   $form->handleRequest($request);
13
14   if ($form->isSubmitted() && $form->isValid())
15   {
16    $em = $this->getDoctrine()->getManager();
17    $em->persist($category);
18    $em->flush();
19    return $this->redirectToRoute('categories_show',
      ↪  array('id' => $category->getId()));
20   }
21
22   return $this->render('categories/new.html.twig', array(
23    'category' => $category,
24    'form' => $form->createView(),
25    ));
26 }
```

— Ligne 6 : permet de spécifier de quelle manière peut être appelée la route. Ici deux choix sont possible : en GET, c'est à dire par une URL, ou en POST, lors de la soumission du formulaire.
— Ligne 8 : l'objet $request en paramètre permet de récupérer tout ce qui est envoyé par le navigateur du client.
— Ligne 10 : création d'un nouvel objet de type Catégorie. A noter que l'on pourrait passer des paramètres dans le constructeur.
— Ligne 11 : création d'un objet de type form. Cet objet va contenir le formulaire qui sera affiché. La création d'un formulaire nécessite au moins deux paramètres : un formType (*cf.* 3.2), et un objet correspondant au type pour lequel le formulaire est destiné.
— Ligne 12 : cette instruction permet de faire coïncider les éléments issus du navigateur avec les données attendues dans le formulaire précédemment initialisé. Lors de l'appel en GET cette instruction n'effectue pas grand chose. Par contre en post, elle recopie les données du navigateur dans les propriétés de l'objet.
— Ligne 14 : ce test permet de regarder si le formulaire a été soumis (méthode POST) et si les données saisies sont valides avec le descriptif donné dans le formType. Si rien n'est précisé sur le type de données, ce test vérifie juste si les champs obligatoires sont bien saisis.
— Ligne 16 : Cette instruction permet de récupérer la connexion à la base de données.
— Ligne 17 : cette instruction permet de persister l'objet dans la file d'attente de la base de données. A ce stade $category contient les informations saisies par l'utilisateur. On pourrait faire des manipulations/modifications sur l'objet avant de le persister.
— Ligne 18 : cette ligne écrit les modifications (la file d'attente) dans la base de données.
— Ligne 19 : l'objet a bien été ajouté dans la base de données, l'application redirige vers l'affichage détaillé de l'objet créé.
— Ligne 22 : dans le cas d'un appel en GET et sans soumission du formulaire, la vue contenant l'affichage du formulaire est renvoyée (*cf.* 3.3.1). Noter la présence d'une clé 'form' qui contient la vue générée par le formulaire (createView)

3.1.4 La méthode show

Cette méthode permet l'affiche des données d'un objet précis. Dans cette page on trouve aussi un bouton modifier et un bouton supprimer.

```php
<?php
/**
* Finds and displays a Categories entity.
*
* @Route("/{id}", name="categories_show")
* @Method("GET")
*/
public function showAction(Categories $category)
{
  $deleteForm = $this->createDeleteForm($category);

  return $this->render('categories/show.html.twig', array(
        'category' => $category,
        'delete_form' => $deleteForm->createView(),
        ));
}
```

- Ligne 5 : la route déclare prendre un paramètre (ici id).
- Ligne 8 : l'en-tête de la méthode prendre donc un paramètre. Ici il n'y a qu'un seul paramètre, le nommage n'est pas gênant. Dans le cas de plusieurs paramètre les noms doivent concorder. A noter qu'on précise une entité (Catégorie) alors qu'on récupère un id dans la route. Cette ecrite s'appelle du "paramConverter" c'est à dire que Symfony exécute une requête permettant d'aller chercher dans la table Catégories un objet ayant l'ID passé en paramètre. Cela évite d'avoir à écrire une requête (find) dans le contrôleur.
- Ligne 10 : cette ligne permet de créer un bouton de formulaire pour la suppression. Passer par un bouton de formulaire pour une suppression est considéré comme plus sécurisée. De plus elle permet de filtrer la méthode DELETE lors de l'appel d'une route.
- Ligne 12 : cette ligne permet de renvoyer une vue, avec l'objet à afficher et le formulaire pour le bouton de suppression.

3.1.5 La méthode edit

Cette méthode permet de générer un formulaire pré-rempli pour la modification d'un objet en particulier. En y regardant plus en détail, on remarque que finalement cette méthode est un mélange de la méthode new (Requet, création d'un formulaire, récupération de données, mise à jour de la base de données) et de la méthode show (récupération d'un objet précis en URL)

```php
<?php
/**
* Displays a form to edit an existing Categories entity.
*
* @Route("/{id}/edit", name="categories_edit")
* @Method({"GET", "POST"})
*/
public function editAction(Request $request, Categories
↪  $category)
{
  $deleteForm = $this->createDeleteForm($category);
  $editForm =
  ↪  $this->createForm('AppBundle\Form\CategoriesType',
  ↪  $category);
  $editForm->handleRequest($request);

  if ($editForm->isSubmitted() && $editForm->isValid())
  {
        $em = $this->getDoctrine()->getManager();
        $em->persist($category);
        $em->flush();

        return $this->redirectToRoute('categories_edit',
        ↪  array('id' => $category->getId()));
  }

  return $this->render('categories/edit.html.twig', array(
        'category' => $category,
        'edit_form' => $editForm->createView(),
        'delete_form' => $deleteForm->createView(),
        ));
}
```

Voir les codes précédents pour les explications des différentes lignes. A noter, toutefois, que l'update et l'insert se déroule avec la même instruction (persist). Doctrine fait ensuite la part des choses pour savoir quelle requête il doit exécuter.

3.1.6 La méthode delete

Cette méthode permet de supprimer un élément de la base de données. Cette page est une nouvelle fois un mélange entre le new (création d'un formulaire pour le bouton delete) et show (récupération d'un objet en particulier).

```php
<?php
/**
* Deletes a Categories entity.
*
* @Route("/{id}", name="categories_delete")
* @Method("DELETE")
*/
public function deleteAction(Request $request, Categories
↪  $category)
{
  $form = $this->createDeleteForm($category);
  $form->handleRequest($request);

  if ($form->isSubmitted() && $form->isValid())
  {
        $em = $this->getDoctrine()->getManager();
        $em->remove($category);
        $em->flush();
  }

  return $this->redirectToRoute('categories_index');
}
```

Voir les exemples précédent pour le détail du code. A noter la ligne 16 qui correspond à une requête delete. L'effacement n'est réellement fait que lors du flush() de la ligne suivante.

3.1.7 La méthode privée "delete form"

Cette méthode permet de créer le formulaire de suppression. A noter que ce bouton étant utilisé plusieurs fois, il est écrit dans une méthode private (donc non accessible par une URL, d'ailleurs il n'y a pas de route). Cette méthode permet de créer un formulaire qui possède une action qui est une route avec un paramètre et un bouton qui enverra les éléments avec une requête DELETE. Vous voyez dans cette exemple comment créer un formulaire sans passer par un fichier formType.

```php
<?php
/**
* Creates a form to delete a Categories entity.
*
* @param Categories $category The Categories entity
```

```
6   *
7   * @return \Symfony\Component\Form\Form The form
8   */
9   private function createDeleteForm(Categories $category)
10  {
11    return $this->createFormBuilder()
12      ->setAction($this->generateUrl('categories_delete',
        ↪  array('id' => $category->getId())))
13      ->setMethod('DELETE')
14      ->getForm();
15  }
```

3.2 Le FormType

3.2.1 Principe

Un formulaire est un élément essentiel d'un site web ou d'une application. Il va permettre de recueillir des données provenant de l'utilisateur. Écrire un formulaire est souvent fastidieux et rébarbatif. Et modifier l'apparence des formulaires lors d'une refonte d'un site web peut également être une tâche périlleuse.

Un formulaire va de plus généralement être utilisé deux fois. Une première fois pour la création, et une seconde fois pour la modification. Certes cela n'a rien d'obligatoire, mais ca reste le comportement courant d'un site web.

Dans l'idée de ne jamais écrire plus d'une fois un même morceau de code, Symfony propose la mise en place de fichiers qui décrivent le comportement et l'affichage d'un formulaire. Ce type de fichier, appelé formType peut être généré automatiquement grâce à la console par rapport à une entité précise, on peut avoir plusieurs formType pour une même entité, et on peut aussi avoir des formType qui seraient liés à plusieurs entités, ... Toutes les solutions sont possible.

La ligne ci-dessous permet de générer un formulaire pour une entité précise.

```
1   php bin/console doctrine:generate:form AppBundle:Entity
```

L'utilisation du CRUD génère automatiquement un fichier formType. Dans les deux cas ces fichiers sont très basiques dans leur contenu, et n'exploite pas toutes les fonctionnalités des formulaires.

3.2.2 Exemple

```php
<?php
namespace AppBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;

class CategoriesType extends AbstractType
{
  /**
  * @param FormBuilderInterface $builder
  * @param array $options
  */
  public function buildForm(FormBuilderInterface $builder,
  ↪ array $options)
  {
    $builder->add('libelle')
    ;
  }

  /**
  * @param OptionsResolver $resolver
  */
  public function configureOptions(OptionsResolver
  ↪ $resolver)
  {
     $resolver->setDefaults(array(
       'data_class' => 'AppBundle\Entity\Categories'
     ));
  }
}
```

Le code ci-dessus se décompose en 2 méthodes qui sont regroupées au sein d'une classe héritant de AbstractType qui contient les outils pour décrire un formulaire.

La première méthode : *buildForm* est une énumération des champs qui vont composer le formulaire. Ces champs sont ajoutés dans la variable $builder.

La seconde méthode : *configureOptions* permet de gérer les options de configuration. Le code minimal est le fait de spécifier la "data_class" qui fait

le lien avec l'entité.

3.2.3 Détails de la méthode buildForm

Dans cette méthode, on ajoute à l'objet $builder les champs que l'on souhaite avoir dans le formulaire. Ces champs doivent exister dans l'entité. C'est ce qu'on appelle des champs "mappés". S'ils n'existent pas, il faut le préciser.

Pour ajouter un champ au builder, on utilise la syntaxe :

```
$builder->add('nom_du_champ', type, array(options))
    ->add(...)
    ;
```

Le nom du champ est le nom du paramètre dans l'entité. Le type de champ peut être déterminé automatiquement en fonction du type dans l'entité. Cependant il est conseillé de le définir de manière explicite. Symfony propose de nombreux types : `http://symfony.com/doc/current/forms.html`, chapitre "Built-in Field Types". Les options sont également nombreuses. Certains sont communes (label, class, required, ...), d'autres sont spécifiques au type choisi (voir la documentation précédente).

Symfony essaye de deviner les options en fonction de l'entité (label, size, required, ...).

3.3 Les vues

3.3.1 Vue new

Le fichier ci-dessous représente le code minimale pour la vue de création d'un objet. Elle hérite de la vue de base, et insére un formulaire.

```
{% extends 'base.html.twig' %}

{% block body %}
<h1>Categories creation</h1>

{{ form_start(form) }}
{{ form_widget(form) }}
<input type="submit" value="Create" />
{{ form_end(form) }}

<ul>
```

```
12      <li>
13          <a href="{{ path('categories_index') }}">Back to the
    ↪  list</a>
14      </li>
15  </ul>
16  {% endblock %}
```

— Ligne 1 ; héritage du formulaire de base
— Ligne 3 : insertion du code dans le block body du template de base.
— Ligne 6 : création du formulaire. La variable form correspond à l'objet $form créé dans le contrôleur et passé dans le tableau à la vue, via la clé 'form'.
— Ligne 7 : cette ligne insère tous les champs créés dans le formType.
— Ligne 8 : cette ligne ajoute le bouton de validation du formulaire. En procédant ainsi, on peut utiliser le même formulaire pour un ajout ou une modification.
— Ligne 9 : affichage de la fin du formulaire (fermeture de la balise, insertion des champs masqués, CSRF...)

Cette solution affiche le formulaire de manière brut. Il est bien sûr possible de personnaliser le rendu des formulaire de manière globale afin de les intégrer dans le projet. Cela se passe dans les thèmes de formulaire. Toutes les informations se trouvent dans la documentation officielle de Symfony : `https://symfony.com/doc/current/form/form_themes.html`