



Doctrine

1. Présentation du travail

L'idée est de créer une base de données à partir d'un diagramme de classes.

Après avoir chargé un jeu d'essai, vous manipulerez un peu les données dans une interface

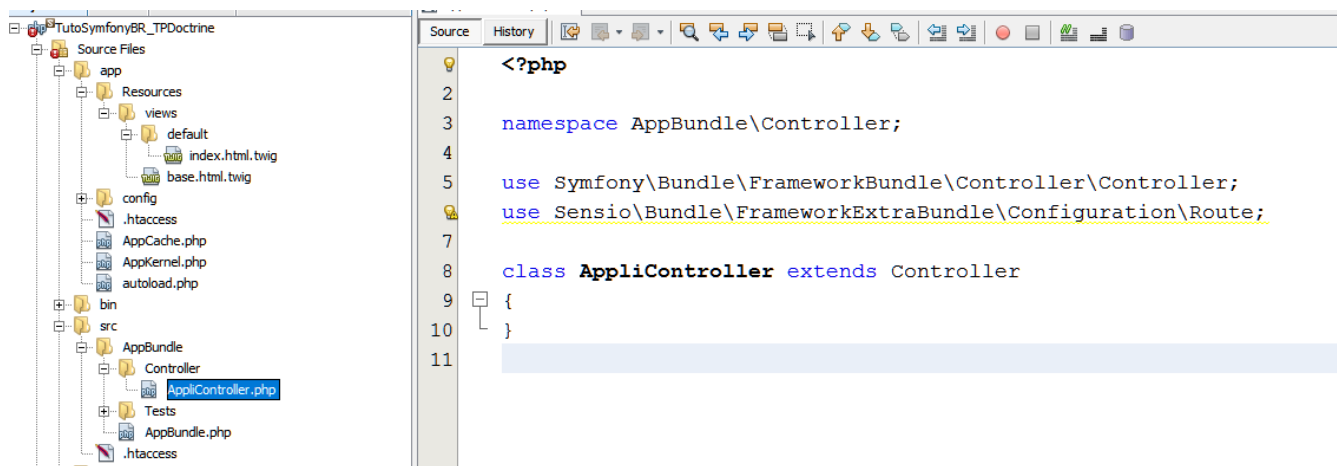
2. Préparation du travail

- ✓ Créer le projet TutoSymfonyBR_TPDoctrine
- ✓ Vous utiliserez le bundle AppBundle
- ✓ Créer un virtualHost si ce n'est déjà fait. Ici ce sera symfony.br

Faire un test avec l'URL :

http://symfony.br/TutoSymfonyBR_TPDoctrine/web/app_dev.php

- ✓ Supprimer le contrôleur DefaultController
- ✓ Créer le contrôleur AppliController (classe AppliController, fichier AppliController.php). Pas d'action pour l'instant. Routing en annotations



Partie 1 : la structure

On va commencer par saisir les paramètres de la base de données. Pour ceci, allez dans le fichier parameters.yml :

T:\Wampsites\Symfony\TutoSymfonyBR_Doctrine\app\config\parameters.yml

Les paramètres qui nous intéressent sont :

- ✓ database_host: 127.0.0.1
- ✓ database_port: null

- ✓ database_name: TPSymfonyDoctrine
- ✓ database_user: usymfony
- ✓ database_password: usymfony

Vous aurez au préalable créé l'utilisateur symfony/symfony qui pour l'instant aura le droit de créer des bases de données. Vous créerez cet utilisateur sous le compte de root

```
parameters:
    database_host: 127.0.0.1
    database_port: 3306
    database_name: TPSymfonyDoctrine
    database_user: symfony
    database_password: symfony
    mailer_transport: smtp
    mailer_host: 127.0.0.1
    mailer_user: null
    mailer_password: null
    secret: 3742c175b8c0c4ca4a3b8d89761b6da9f308b32b
```

Details for account newuser@%

Login	Account Limits	Administrative Roles	Schema Privileges																						
<table border="1"> <thead> <tr> <th>Role</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td><input type="checkbox"/> DBA</td> <td>grants the rights to per</td> </tr> <tr> <td><input type="checkbox"/> MaintenanceAdmin</td> <td>grants rights needed to</td> </tr> <tr> <td><input type="checkbox"/> ProcessAdmin</td> <td>rights needed to asses</td> </tr> <tr> <td><input type="checkbox"/> UserAdmin</td> <td>grants rights to create u</td> </tr> <tr> <td><input type="checkbox"/> SecurityAdmin</td> <td>rights to manage login</td> </tr> <tr> <td><input type="checkbox"/> MonitorAdmin</td> <td>minimum set of rights r</td> </tr> <tr> <td><input type="checkbox"/> DBManager</td> <td>grants full rights on all</td> </tr> <tr> <td><input checked="" type="checkbox"/> DBDesigner</td> <td>rights to create and rev</td> </tr> <tr> <td><input type="checkbox"/> ReplicationAdmin</td> <td>rights needed to setup</td> </tr> <tr> <td><input type="checkbox"/> BackupAdmin</td> <td>minimal rights needed</td> </tr> </tbody> </table>				Role	Description	<input type="checkbox"/> DBA	grants the rights to per	<input type="checkbox"/> MaintenanceAdmin	grants rights needed to	<input type="checkbox"/> ProcessAdmin	rights needed to asses	<input type="checkbox"/> UserAdmin	grants rights to create u	<input type="checkbox"/> SecurityAdmin	rights to manage login	<input type="checkbox"/> MonitorAdmin	minimum set of rights r	<input type="checkbox"/> DBManager	grants full rights on all	<input checked="" type="checkbox"/> DBDesigner	rights to create and rev	<input type="checkbox"/> ReplicationAdmin	rights needed to setup	<input type="checkbox"/> BackupAdmin	minimal rights needed
Role	Description																								
<input type="checkbox"/> DBA	grants the rights to per																								
<input type="checkbox"/> MaintenanceAdmin	grants rights needed to																								
<input type="checkbox"/> ProcessAdmin	rights needed to asses																								
<input type="checkbox"/> UserAdmin	grants rights to create u																								
<input type="checkbox"/> SecurityAdmin	rights to manage login																								
<input type="checkbox"/> MonitorAdmin	minimum set of rights r																								
<input type="checkbox"/> DBManager	grants full rights on all																								
<input checked="" type="checkbox"/> DBDesigner	rights to create and rev																								
<input type="checkbox"/> ReplicationAdmin	rights needed to setup																								
<input type="checkbox"/> BackupAdmin	minimal rights needed																								

On donne l'adresse du serveur, le user, le pwd,... mais on ne donne pas d'indication sur le SGBD Est-ce le hasard si c'est mysql ?

Vous trouverez la solution dans le fichier\TutoSymfonyBR_Doctrine\app\config\config.yml

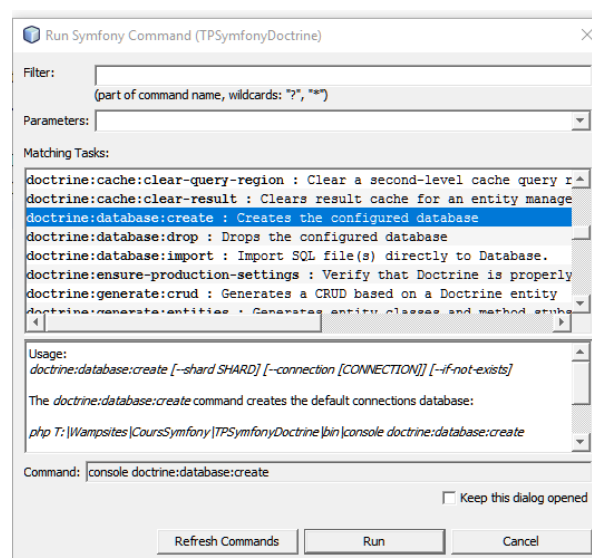
```
# Doctrine Configuration
doctrine:
  dbal:
    driver:   pdo_mysql
    host:     "%database_host%"
    port:     "%database_port%"
    dbname:   "%database_name%"
    user:     "%database_user%"
    password: "%database_password%"
    charset:  UTF8
    # if using pdo_sqlite as your database driver:
    #  1. add the path in parameters.yml
    #     e.g. database_path: "%kernel.root_dir%/data/data.db3"
    #  2. Uncomment database_path in parameters.yml.dist
    #  3. Uncomment next line:
    #     path:   "%database_path%"

orm:
```

On remarque que le driver est indiqué en dur ... pdo_mysql. Les autres paramètres seront mappés sur les paramètres correspondants du fichier config.yml.

Par exemple host: "%database_host%" récupèrera la valeur du paramètre database_host: 127.0.0.1 du fichier parameters.yml

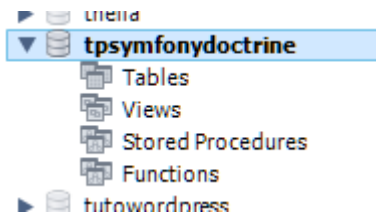
Vous allez créer la base de données TPSymfonyDoctrine à l'aide de la commande Symfony doctrine:database:create



La base de données va donc se créer.

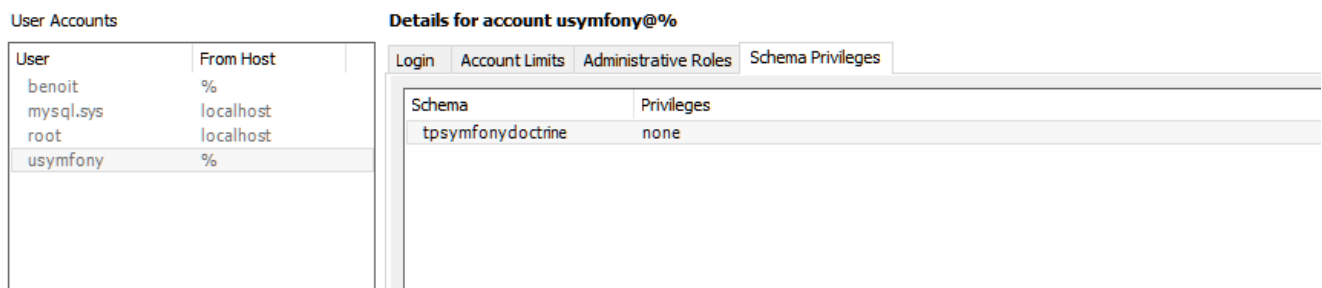
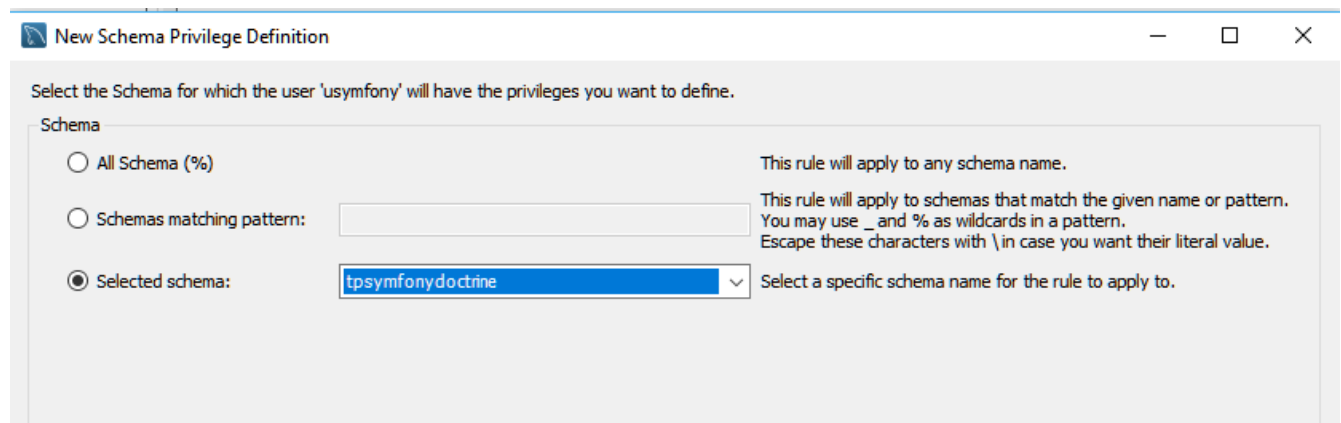
```
Symfony 3 (TutoSymfonyBR_TPDoctrine) x | Symfony 3 (TPSymfonyDoctrine) x |
"C:\wamp64\bin\php\php7.0.10\php.exe" "T:\Wampsites\CoursSymfony\TPSymfonyDoctrine\bin\console" "--ansi" "doctrine:database:create"
Created database 'TPSymfonyDoctrine' for connection named default
Done.
```

On va le vérifier avec le client mysql workbench :

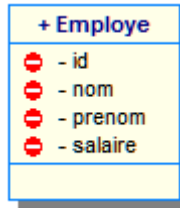


La base a bien été créée.

Vous allez maintenant donner à l'utilisateur usymfony tous les privilèges sur cette base :
Dans Mysqlworkbench



Vous allez maintenant créer l'Entity Employe qui correspond à cette classe :

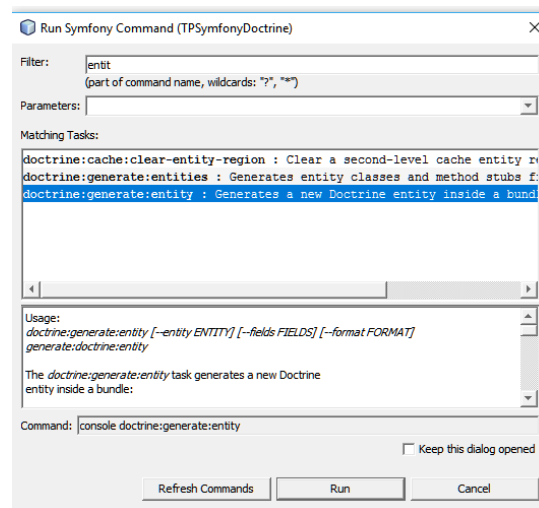


id est un entier, nom et prénom des chaines de caractères et salaire un décimal

Pour faire ceci, ouvrez une console de commande Symfony et appelez la commande Doctrine:generate:entity

qui va nous guider dans la création de la classe Employe.

N'oubliez pas de préciser en paramètre le bundle concerné et le nom de l'Entity (classe) à créer.



La première chose à faire est de confirmer le nom de l'entity : vous validez

```
"C:\wamp64\bin\php\php7.0.10\php.exe" "T:\Wampsites\CoursSymfony\TPSymfonyDoctrine\bin\console" "--ansi" "doctrine:generate:entity"
```

```
Welcome to the Doctrine2 entity generator
```

```
This command helps you generate Doctrine2 entities.
```

```
First, you need to give the entity name you want to generate.
```

```
You must use the shortcut notation like AcmeBlogBundle:Post.
```

```
The Entity shortcut name:
```

```
AppBundle:Employe|
```

AppEnsuite le format des informations de mapping : vous choisirez l'option par défaut : annotation

Determine the format to use for the mapping information.

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

Vous devrez ensuite renseigner les différents champs.



SAUF le champ id qui sera créé automatiquement. Le premier champ à créer est donc le nom

En lisant le message, vous découvrirez les différents types disponibles (attention à la casse). Pour le nom, le type sera string, qui est le type par défaut et la longueur du champ, maximum 255. Saisir 50.

```
New field name (press <return> to stop adding fields):
Instead of starting with a blank entity, you can add some fields now.
Note that the primary key will be added automatically (named id).

Available types: array, simple_array, json_array, object,
boolean, integer, smallint, bigint, string, text, datetime, datetimetz,
date, time, decimal, float, binary, blob, guid.

nom|
```

```
New field name (press <return> to stop adding fields): nom
Field type [string]:
Field length [255]: 50

New field name (press <return> to stop adding fields): |
```

Vous pouvez ensuite arrêter la saisie des champs en faisant *enter* tout simplement, ou continuer en saisissant un nouveau nom d'attribut (champ)

Les valeurs pour l'attribut nom :

```
nom
Field type [string]:
Field length [255]: 50
Is nullable [false]:
Unique [false]:
New field name (press <return> to stop adding fields):
```

Continuez avec le prénom (sans accent) et le salaire. Vous devriez obtenir ceci :

Il ne vous reste plus qu'à valider.

Vous devrez répondre à la question suivante ...

```
Do you want to generate an empty repository class [no]? |
```

Vous répondrez par oui (yes). En effet, comme vu en cours, le repository va permettre de faire le lien entre l'entity et la base de données. Dans le sens BD vers entity, le repository va hydrater les objets, c'est à dire qu'il va créer et valoriser les objets et collections à l'aide des données extraites de la base.

Le repository de l'entity sera une classe qui sera chargée d'extraire les données de l'entity. L'équivalent du select sur une table de BD. Cette méthode permet de découpler les données de la classe (entity) et les traitements qu'on peut faire sur les objets de l'entity.

```
prenom
Field type [string]:
Field length [255]: 50
Is nullable [false]:
Unique [false]:
New field name (press <return> to stop adding fields):
salaire
Field type [string]: decimal
Precision [10]: 8
Scale: 2
Is nullable [false]:
Unique [false]:
New field name (press <return> to stop adding fields):
.
```

Validez (*Enter*) pour terminer la saisie des attributs

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

```
Entity generation
```

```
created .\src\AppBundle\Entity/
created .\src\AppBundle\Entity\Employee.php
> Generating entity class T:\Wampsites\CoursSymfony\TPSymfonyDoctrine\src\AppBundle\Entity\Employee.php: OK!
> Generating repository class T:\Wampsites\CoursSymfony\TPSymfonyDoctrine\src\AppBundle\Repository\EmployeeRepository.php: OK!
```

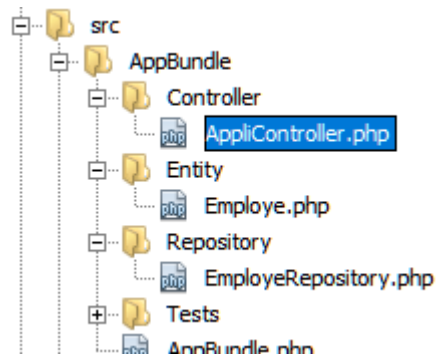
```
Everything is OK! Now get to work :).
```

```
Done.
```



Que s'est-il passé au niveau de l'application ?
Que s'est-il passé au niveau de la BD ?

Au niveau de l'application, on remarque 2 nouveaux dossiers Entity et Repository dans le dossier de notre bundle



Et si on ouvre l'arborescence, on voit 2 fichiers créés :

- ✓ Dans le dossier Entity : Employe.php (classe Employe)
- ✓ Dans la dossier Repository : EmployeRepository.php

En ouvrant le fichier Employee.php, on découvre la classe Employee créée et dans les annotations, tous les renseignements saisis à l'étape précédente :

```
<?php

namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Employee
 *
 * @ORM\Table(name="employee")
 * @ORM\Entity(repositoryClass="AppBundle\Repository\EmployeeRepository")
 */
class Employee
{
    /**
     * @var int
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string
     *
     * @ORM\Column(name="nom", type="string", length=50)
     */
    private $nom;
```

On voit surtout que l'attribut id a été généré !!!

L'annotation



```
* @ORM\Id
* @ORM\GeneratedValue(strategy="AUTO")
*/
```

signifie que cette valeur sera auto-incrémentée sous mysql.

On aurait bien entendu pu tout écrire à la main !!!

Dans la classe EmployeeRepository :

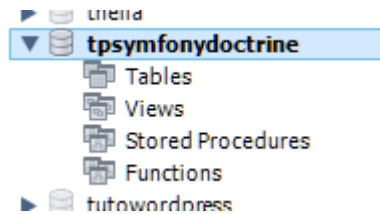
```
<?php
```

```
namespace AppBundle\Repository;
```

```
/**
 * EmployeRepository
 *
 * This class was generated by the Doctrine ORM. Add your own custom
 * repository methods below.
 */
class EmployeRepository extends \Doctrine\ORM\EntityRepository
{
}
```

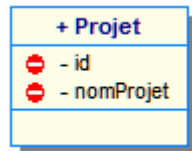
On la complètera ultérieurement, elle comprendra la partie traitements de la classe Employe.

Voyons côté BD :



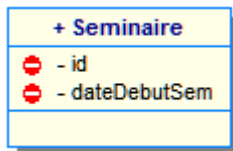
La base tpsymfonydoctrine ne contient encore aucune table. C'est normal, on n'a pas demandé à Doctrine de mettre à k=jour ce schéma (cette base).

Il vous est maintenant demandé de créer sur le même modèle les classes



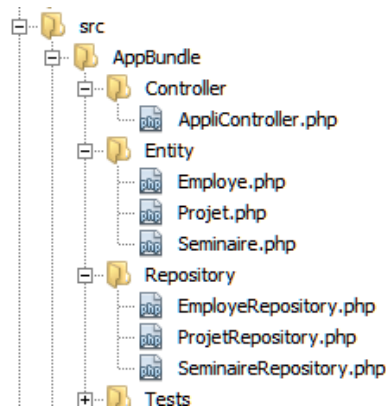
id : sera généré automatiquement
nomProjet : chaîne de 50 caractères

et



id : sera généré automatiquement
dateDebutSem format date

Vous devriez maintenant avoir les classes suivantes dans les dossiers Entity et Repository de votre bundle AppBundle :



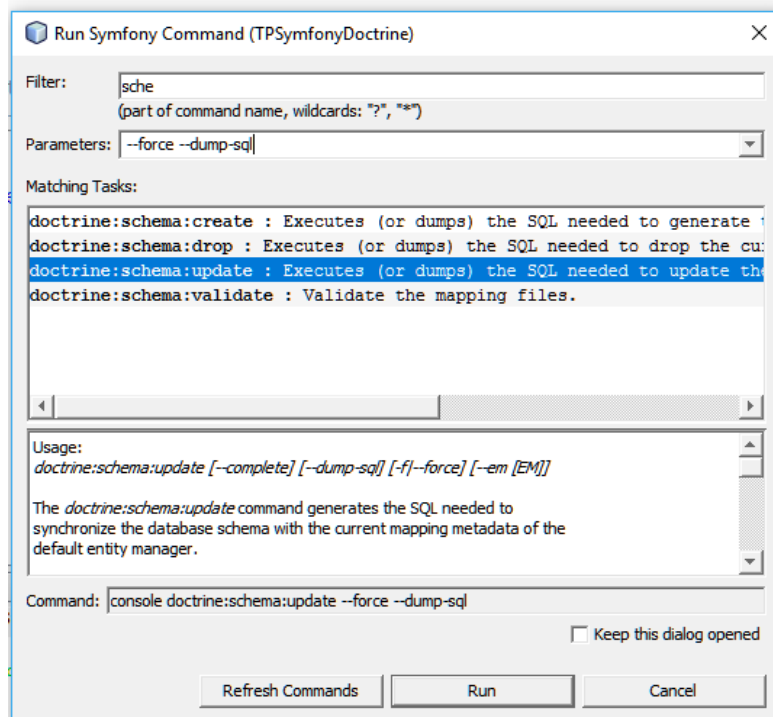
Je vous conseille d'ouvrir les fichiers Entity pour vérifier les informations contenues. Sont-elles conformes à ce qui est demandé ?

On va maintenant mettre à jour le schéma de la base de données.
Pour ceci, on va utiliser la commande symfony doctrine:schema:update

Avec les bonnes options ...

--dump-sql : pour générer les ordres sql de mise à jour du schéma,

--force : pour que ces ordres soient exécutés....



On pourra faire 2 étapes pour cette phase, le dump d'abord et le force ensuite ...

Dans la fenêtre de résultat on voit bien les ordres SQL générés :

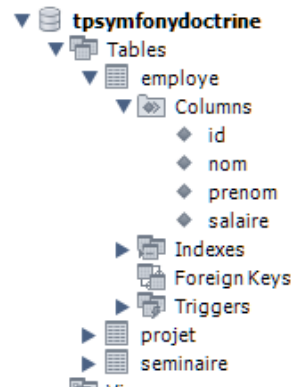
```

ch Results | Output X
Symfony 3 (TutoSymfonyBR_TPDoctrine) X | Symfony 3 (TPSymfonyDoctrine) X |
"C:\wamp64\bin\php\php7.0.10\php.exe" "T:\Wampsites\CoursSymfony\TPSymfonyDoctrine\bin\console" "--ansi" "doctrine:schema:update" "--force" "--dump-sql"
CREATE TABLE employe (id INT AUTO_INCREMENT NOT NULL, nom VARCHAR(50) NOT NULL, prenom VARCHAR(50) NOT NULL, salaire NUMERIC(8, 2) NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci ENGINE = InnoDB;
CREATE TABLE projet (id INT AUTO_INCREMENT NOT NULL, nomProjet VARCHAR(50) NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci ENGINE = InnoDB;
CREATE TABLE seminaire (id INT AUTO_INCREMENT NOT NULL, dateDebutSem DATE NOT NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8 COLLATE utf8_unicode_ci ENGINE = InnoDB;

Updating database schema...
Database schema updated successfully! "3" queries were executed
Done.
|

```

Il ne reste plus qu'à aller voir dans la base ce qu'il s'est passé :



Les tables ont bien été créées.

Dans un projet, la BD est amenée à évoluer ... on va maintenant rajouter l'attribut ville à l'entity Employe.

Pas de commande symfony, on va donc créer l'attribut ville sur 40 caractères alphanumériques et avec une valeur par défaut : Toulon.

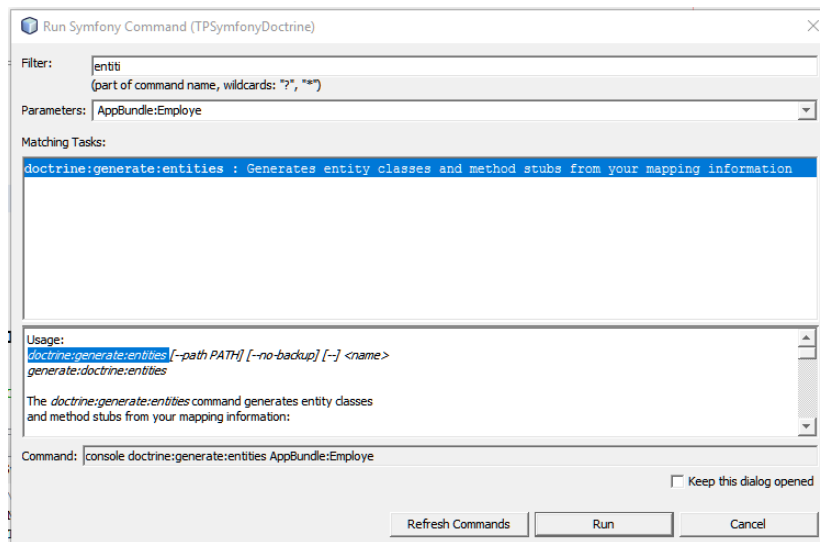
Modifiez l'entity Employe : en rajoutant l'attribut ville et ses getter/setter :

```

/**
 * @var string
 *
 * @ORM\Column(name="ville", type="string", length=40 ,options={"default"="Toulon"})
 */
private $ville;

```

Pour générer les accesseurs et mutateurs, vous utiliserez la commande : `doctrine:generate:entities`



```
Symfony 3 (TutoSymfonyBR_TPDOctrine) x | Symfony 3 (TPSymfonyDoctrine) x |
"C:\wamp64\bin\php\php7.0.10\php.exe" "T:\Wampsites\CoursSymfony\TPSymfonyDoc
Generating entity "AppBundle\Entity\Employe"
> backing up Employe.php to Employe.php~
> generating AppBundle\Entity\Employe
Done.
```

Voyons ce qui a été généré au niveau du code :

```
public function setVille($ville)
{
    $this->ville = $ville;

    return $this;
}

/**
 * Get ville
 *
 * @return string
 */
public function getVille()
{
    return $this->ville;
}
```

Vous mettrez bien entendu la base à jour doctrine:schema:update --dump-sql --force
... après avoir enregistré vos modifications ...



Et constaterez les ordres sql générés :

Seules les modifications ont été prises en compte !!!

```
h Results | Output × |
Symfony 3 (TutoSymfonyBR_TPDoctrine) × | Symfony 3 (TPSymfonyDoctrine) × |
"C:\wamp64\bin\php\php7.0.10\php.exe" "T:\Wampsites\CoursSymfony\TPSymfonyDoctrine\bin\console" "--ansi" "doctri
ALTER TABLE employe ADD ville VARCHAR(40) NOT NULL;

Updating database schema...
Database schema updated successfully! "1" query was executed
Done.
```

Oups Les index vous vous rappelez ce qu'est un index ?

On va rajouter un index :

- ✓ Sur la colonne nom de l'Entity Employe
- ✓ Sur la colonne ville de l'entity Employe

Tout ceci se passe par annotation au niveau de la classe :

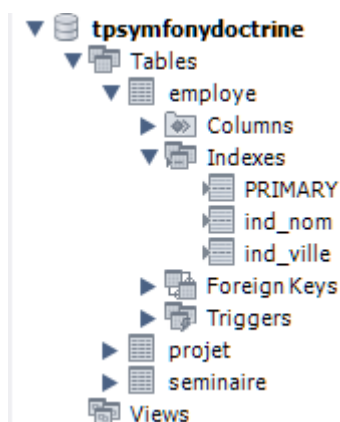
```
/**
 * Employe
 *
 * @ORM\Table(name="employe" , indexes={@ORM\Index(name="ind_nom", columns={"nom"}),
 * @ORM\Index(name="ind_ville", columns={"ville"})})
 * @ORM\Entity(repositoryClass="AppBundle\Repository\EmployeRepository")
 */
```

Et on met à jour la base : doctrine:schema:update --dump-sql --force

```
h Results | Output - Symfony 3 (TPSymfonyDoctrine) × |
"C:\wamp64\bin\php\php7.0.10\php.exe" "T:\Wampsites\CoursSymfony\TPSymfonyDoctrine\bin\console" "--ansi" "doctr:
CREATE INDEX ind_nom ON employe (nom);
CREATE INDEX ind_ville ON employe (ville);

Updating database schema...
Database schema updated successfully! "2" queries were executed
Done.
```

Et on vérifie dans la BD :



Tout a bien fonctionné !!!!



On remarque l'index *PRIMARY* qui a été créé automatiquement avec la contrainte de *PRIMARY KEY*

Ce n'est pas fini on va maintenant établir les liens entre les entity

On sait que :

- ✓ Un employé travaille sur UN projet et que sur UN projet travaillent N employés.

On va donc faire le choix de créer une relation *ManyToOne* à partir de l'entity employé car plusieurs employés travaillent sur UN projet. On se trouvera donc dans l'entity *Employe* avec un attribut *\$projet* !

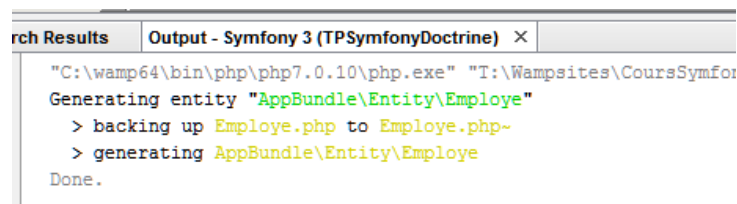
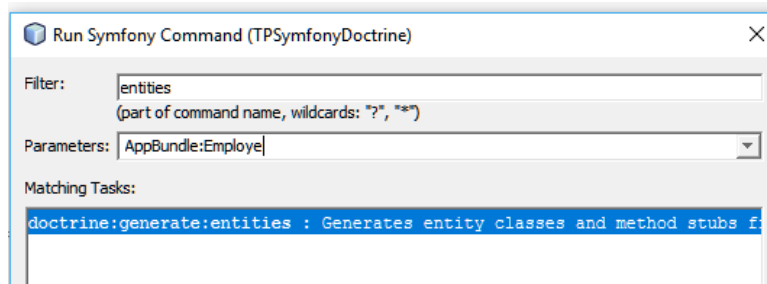


Attention en Doctrine, on fonctionne comme en UML ... à l'envers !!!
On ne peut pas créer une relation *OneToMany* de employe à projet.
On aurait pu créer une relation *OneToMany* de projet vers employe.

Pour faire ceci, on va créer un attribut *\$projet* dans l'entity *Employe* et indiquer la relation à l'entity *Projet* (du bundle) par annotation :

```
private $ville;  
  
/**  
 * @ORM\ManyToOne(targetEntity="\AppBundle\Entity\Projet")  
 */  
private $projet;
```

.... Et on va en profiter pour rajouter les getter/setter, toujours avec la commande *doctrine:generate:entities*



```

/**
 * Set projet
 *
 * @param \AppBundle\Entity\Projet $projet
 * @return Employe
 */
public function setProjet(\AppBundle\Entity\Projet $projet = null) {
    $this->projet = $projet;

    return $this;
}

/**
 * Get projet
 *
 * @return \AppBundle\Entity\Projet
 */
public function getProjet() {
    return $this->projet;
}

```

... vous remarquerez ici qu'on autorise un employé a ne travailler sur aucun projet !!!

```
ty\Projet $projet = null) {
```



Si le cahier des charges l'exige, on pourrait enlever cette option.

On met à jour la BD, toujours le même principe :

```

"C:\wamp64\bin\php\php7.0.10\php.exe" "T:\Wampsites\CoursSymfony\TPSymfonyDoctrine\bin\console" "--ansi"
ALTER TABLE employe ADD projet_id INT DEFAULT NULL;
ALTER TABLE employe ADD CONSTRAINT FK_F804D3B9C18272 FOREIGN KEY (projet_id) REFERENCES projet (id);
CREATE INDEX IDX_F804D3B9C18272 ON employe (projet_id);

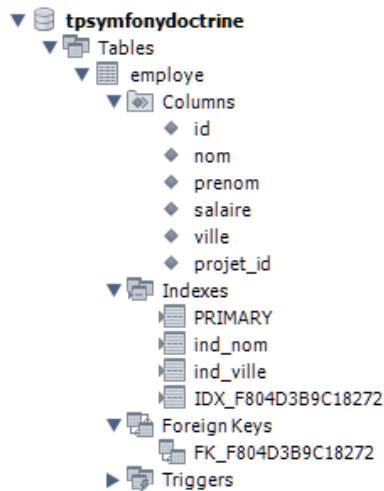
Updating database schema...
Database schema updated successfully! "3" queries were executed
Done.

```



Il n'existe pas une annotation sous doctrine pour modifier le nom des clés étrangères

Et on vérifie la BD :



On constate que :

- ✓ Une colonne a été rajoutée : projet_id
- ✓ Un index a été créé
- ✓ Une clé étrangère a été créée

C'est conforme aux ordres SQL générés.

Il ne nous reste plus maintenant qu'à relier les employés aux séminaires pour gérer les inscriptions

- ✓ UN employé peut s'inscrire à plusieurs séminaires et à un séminaire, peuvent s'inscrire plusieurs employés.

On va donc devoir créer une relation ManyToMany.

L'idée :



créer une collection d'objets \$seminaires dans l'entity Employe

créer une collection d'objets \$employes dans l'entity Seminaire

- ✓ \$seminaires représentera les séminaires auxquels s'est inscrit l'employé
- ✓ \$employes représentera les employés inscrits à un séminaire.

Encore une fois, on va créer les annotationsen soignant les noms à attribuer !!!

Dans l'entity Employe, rajouter l'annotation suivante pour l'attribut \$seminaires :

```

/**
 * @ORM\ManyToMany(targetEntity="AppBundle\Entity\Seminaire", inversedBy="employes")
 * @ORM\JoinTable(
 *     name="Inscrit",
 *     joinColumns={@ORM\JoinColumn(name="idEmploye", referencedColumnName="id")},
 *     inverseJoinColumns={@ORM\JoinColumn(name="IdSeminaire", referencedColumnName="id")}
 * )
 */
private $seminaires;

```

Et l'annotation suivante dans l'entity Seminaire :

```

/**
 * @ORM\ManyToMany(targetEntity="\AppBundle\Entity\Employe", mappedBy="seminaires")
 */
private $employes;

```

Et là, vous pouvez passer un peu de temps à comprendre

Vous pouvez voir dans l'entity Employe :

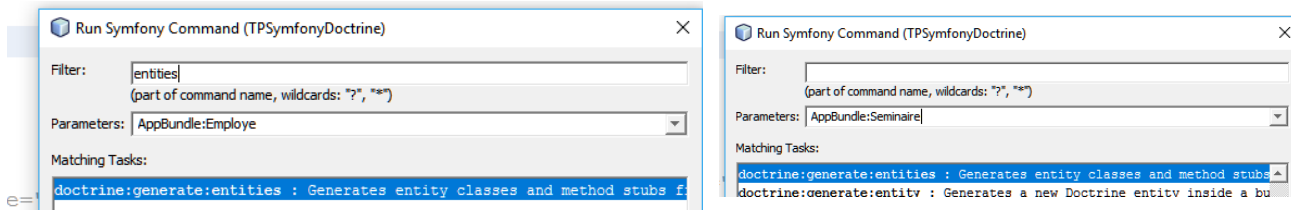
- ✓ Le nom de la table qui va être créée : Inscrit
- ✓ Les colonnes qui seront clé étrangères : idEmploye_id et idSeminaire,
- ✓ L'attribut inversedBy qui représente l'attribut inverse que l'on trouvera dans l'entity Seminaire.

Dans l'entity Seminaire :

- ✓ L'attribut targetEntity qui représente l'entity liée,
- ✓ L'attribut mappedBy qui est l'attribut inverse de l'entity Employe.

C'est tout Enfin, c'est déjà pas mal !!!

Vous pouvez maintenant créer les getters et setters sur les attributs \$employes et \$seminaires avec la commande *doctrine:generate:entities*



Allons regarder un peu notre code dans l'entité Employe et constatons ce qui a été généré :

- Un constructeur : indispensable pour instancier la collection d'objets Seminaires

```

/**
 * Constructor
 */
public function __construct()
{
    $this->seminaires = new \Doctrine\Common\Collections\ArrayCollection();
}

```

Mais encore :

Les méthodes addSeminare et RemoveSeminare pour ajouter et supprimer des séminaires dans la collection \$seminaires.

```

/**
 * Add seminaire
 *
 * @param \AppBundle\Entity\Seminare $seminaire
 *
 * @return Employe
 */
public function addSeminare(\AppBundle\Entity\Seminare $seminaire)
{
    $this->seminaires[] = $seminaire;

    return $this;
}

/**
 * Remove seminaire
 *
 * @param \AppBundle\Entity\Seminare $seminaire
 */
public function removeSeminare(\AppBundle\Entity\Seminare $seminaire)
{
    $this->seminaires->removeElement($seminaire);
}

/**
 * Get seminaires
 *
 * @return \Doctrine\Common\Collections\Collection
 */

```

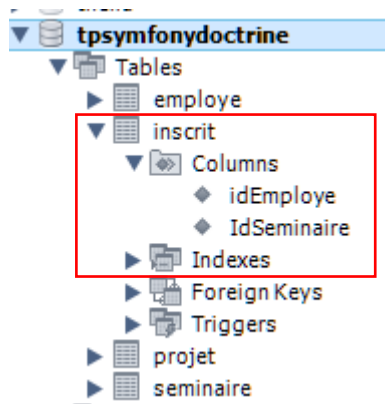
Mettons maintenant à jour notre BD ... comme d'habitude ...

Et le résultat côté symfony :

```
Symfony 3 (TPSymfonyDoctrine) x | Symfony 3 (TutoSymfonyBR_DoctrineV32) x |
"C:\wamp64\bin\php\php7.0.10\php.exe" "T:\Wampsites\CoursSymfony\TPSymfonyDoctrine\bin\console" "--ansi" "doctrine:schema:update" "-
CREATE TABLE Inscrit (idEmploye INT NOT NULL, IdSeminaire INT NOT NULL, INDEX IDX_5DC29AF9E8BDB84B (idEmploye), INDEX IDX_5DC29AF9CJ
ALTER TABLE Inscrit ADD CONSTRAINT FK_5DC29AF9E8BDB84B FOREIGN KEY (idEmploye) REFERENCES employe (id);
ALTER TABLE Inscrit ADD CONSTRAINT FK_5DC29AF9CA8FB511 FOREIGN KEY (IdSeminaire) REFERENCES seminaire (id);

Updating database schema...
Database schema updated successfully! "3" queries were executed
Done
```

Une table a bien été créée....voyons côté BD :



La table *inscrit* a été créée ... vous pouvez vérifier la clé primaire et les clés étrangères !

Voilà pour cette partie, vous avez une idée de la correspondance Entity(classes) < ---- > Table

On va maintenant voir le contenu !!!

Partie 2 : la manipulation du contenu

A l'aide des commandes Symfony, vous allez créer l'entity cours et une relation ManyToOne entre les entities Seminaire et Cours

Entity Cours :

Id (généré automatiquement), libelleCours (string(40)), nbJours (integer)

Dans Seminaire :

```
/**
 * @ORM\ManyToOne(targetEntity="\AppBundle\Entity\Cours")
 * @ORM\JoinColumn(name="idCours", referencedColumnName="id")
 */
private $cours;
```



Remarquez le nom que l'on donne à la clé étrangère : idCours

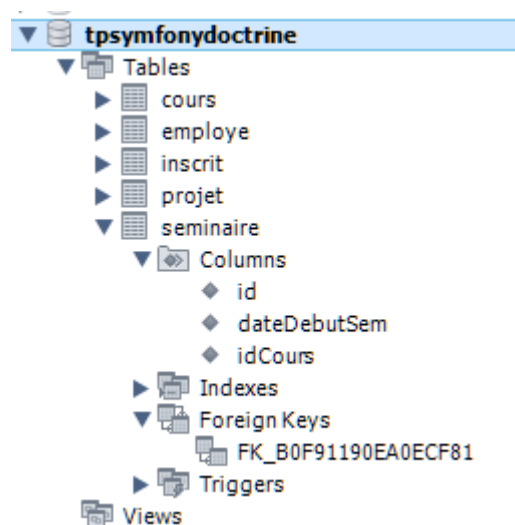
Ne pas oublier de rajouter l'accesseur et le mutateur pour le cours (toujours avec la commande *doctrine:generate:entities*):

```
/**
 * Set cours
 *
 * @param \AppBundle\Entity\Cours $cours
 *
 * @return Seminaire
 */
public function setCours(\AppBundle\Entity\Cours $cours = null)
{
    $this->cours = $cours;

    return $this;
}

/**
 * Get cours
 *
 * @return \AppBundle\Entity\Cours
 */
public function getCours()
{
    return $this->cours;
}
```

On devrait avoir ceci dans la BD après avoir mis à jour le schéma :



Nous voici prêts à démarrer !

On va créer un formulaire Cours

On a vu dans les précédents TP comment générer un formulaire à l'aide d'un objet.
On va le faire ici avec un objet de la classe Cours.

Cet objet sera instancié avec le constructeur par défaut, c'est-à-dire que ses attributs ne contiendront pas de valeur.

On va appeler le formulaire et on va saisir les données du cours sauf son id qui sera généré automatiquement !!! (voir TP Formulaire)

Il ne restera plus dans le contrôleur qu'à récupérer les données saisies (voir TP contrôleur) puis à les persister !

On sait donc présent tout faire ... il n'y a plus qu'à assembler les morceaux !!!

ET voici le code du contrôleur qui va tout faire... sauf vous créer la méthode creerCoursAction :

Attention aux namespaces ... Request et cours ...

La route : /creercours

```

use Symfony\Component\Form\Extension\Core\Type\IntegerType;
use Symfony\Component\Form\Extension\Core\Type\SubmitType;
use AppBundle\Entity\Cours;

class AppliController extends Controller
{
    /**
     * @Route("/creercours")
     */
    public function creerCoursAction(Request $request) {
        $cours = new Cours();
        $formulaire = $this->createFormBuilder($cours)
            ->add('libellecours', TextType::class, array('label' => 'Libellé du cours :'))
            ->add('nbjours', IntegerType::class, array('label' => 'Nombre de jours :'))
            ->add('Enregistrer', SubmitType::class)
            ->getForm();

        $formulaire->handleRequest($request);
        // $this->get('request');
        if ($formulaire->isValid()) {
            // $cours=$formulaire->getData();
            $sem = $this->getDoctrine()->getManager();
            $sem->persist($cours);
            $sem->flush();
            return $this->render('AppBundle:Appli:ok.html.twig', array('message' => "Cours Créé"));
        }
        return $this->render('AppBundle:Appli:cours.html.twig', array('leFormulaire' => $formulaire->createView()));
    }
}

```

La nouveauté :

Si le formulaire renvoyé est valide, on persiste l'objet créé !!!

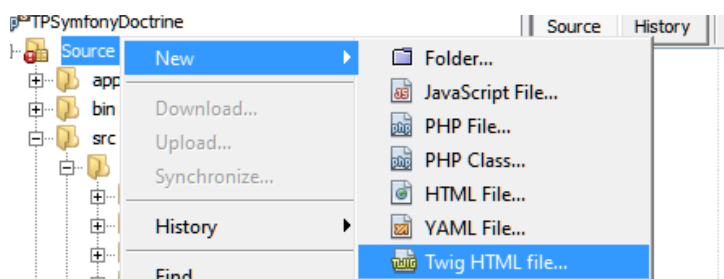
```

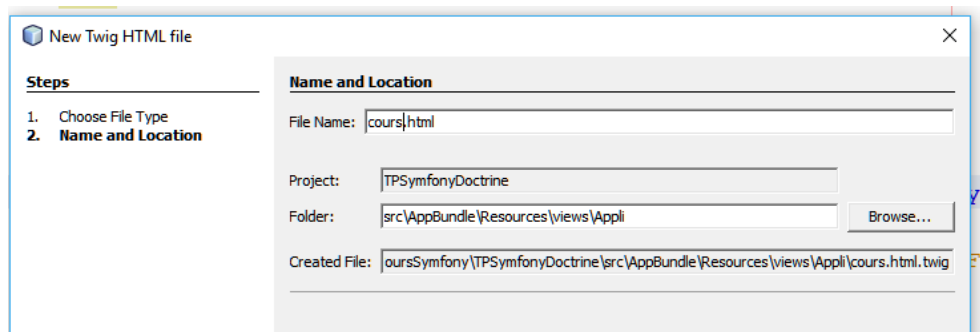
if ($formulaire->isValid()) {
    // $cours=$formulaire->getData();
    $sem = $this->getDoctrine()->getManager();
    $sem->persist($cours);
    $sem->flush();
    return $this->render('BdlsAppliBundle:Appli

```

On crée le formulaire cours.html.twig à l'emplacement :

Attention au dossier et à sa casse !!!





Avec le code suivant :

```
<html>
  <head>
    <title>TODO supply a title</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <div>{{ form_start(leFormulaire) }}
      {{ form_errors(leFormulaire) }}
      {{ form(leFormulaire) }}
    </div>
    {{ form_end(leFormulaire) }}
  </body>
</html>
```

On crée de la même manière le formulaire ok.html.twig dans le même emplacement que cours.html.twig.

Le code :

```
<!DOCTYPE html>
<html>
  <head>
    <title>TODO supply a title</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <div>{{ message }}</div>
  </body>
</html>
```

Et voici le formulaire :

http://symfony.br/TPSymfonyDoctrine/web/app_dev.php/creercours

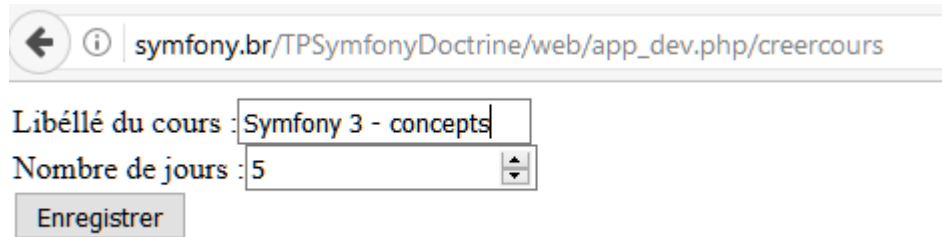


Libellé du cours :

Nombre de jours :

Enregistrer

Que l'on va remplir :

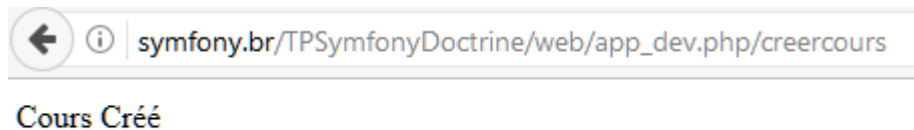


Libellé du cours :

Nombre de jours :

Enregistrer

Un click sur le bouton Enregistrer et l'objet passé en paramètre au formulaire sera hydraté avec les données saisies et Doctrine se chargera de sa persistance :



Cours Créé

Vérifions dans la base de données:

id	libelleCours	nbJours
1	Symfony 3 - concepts	5

L'objet a bien été persisté en enregistrement de la table cours.

3. Le formulaire Séminaire

Créer la route suivante pour arriver au formulaire de saisie d'un nouveau séminaire

On va créer la méthode creerSeminaireAction dans le contrôleur AppliController qui réagit à la route /creerseminaire

Le problème est ...la relation ManyToOne à gérer.

Comment associer 1 cours à 1 séminaire ?

La solution ?

Il existe un type entity dans les champs d'un objet de la classe FormBuilder.

On va donc pouvoir l'inclure dans la création du formulaire.

Syntaxe :

Attention aux namespaces ... séminaire ?

```
/**
 *
 * @Route("/creerseminaire")
 *
 */
public function creerSeminaireAction(Request $request) {
    $seminaire = new Seminaire();
    $formulaire = $this->createFormBuilder($seminaire)
        ->add('datedebutsem', DateTime::class, array('label' => 'Date début :',
            'data' => new \DateTime(),))
        ->add('cours', EntityType::class, array('class' => 'AppBundle:Cours',
            'label' => 'Cours :',
            'required' => 'true',
            'multiple' => false,
            'choice_label' => 'libelleCours',
        ))
        ->add('Enregistrer', SubmitType::class)
        ->getForm();

    $formulaire->handleRequest($request);
    if ($formulaire->isValid()) {
        // $cours = $formulaire->getData();
        $sem = $this->getDoctrine()->getManager();
        $sem->persist($seminaire);
        $sem->flush();
        return $this->render('AppBundle:Appli:ok.html.twig', array('message' => "seminaire Créé"));
    }
    return $this->render('AppBundle:Appli:cours.html.twig', array('leFormulaire' => $formulaire->createView()));
}
```



On vient tout simplement dire au formulaire d'intégrer l'entity Cours et d'afficher dans une liste déroulante l'attribut libellecours de la classe Cours. On a aussi demandé à ce qu'un cours soit choisi.

La date sera initialisée avec la date du jour.

La liste déroulante est automatiquement remplie avec les valeurs des attributs libellecours des cours de l'Entity Cours :

http://symfony.br/TPSymfonyDoctrine/web/app_dev.php/creerseminaire

On n'a plus qu'à choisir la date de début du séminaire et le cours à associer à ce séminaire :

← ⓘ symfony.br/TPSymfonyDoctrine/web/app_dev.php/creerseminaire

Date début :
 Apr ▾ 2 ▾ 2017 ▾
 Cours : Symfony 3 - concepts ▾
 Enregistrer

On enregistre et on voit le résultat :

A l'écran :

← ⓘ symfony.br/TPSymfonyDoctrine/web/app_dev.php/creerseminaire

seminaire Créé

Dans la base de données :

id	dateDebutSem	idCours
1	2017-04-02	1
NULL	NULL	NULL

Le séminaire a bien été créé avec le bon cours associé.



On remarque qu'on utilise la même vue (view) que pour créer le cours (cours.html.twig)

Autre représentation possible :

← → ⓘ symfony.br/TPSymfonyDoctrine/web/app_dev.php/creerseminaire

Date début :
 Apr ▾ 2 ▾ 2017 ▾
 Cours :
☐ Symfony 3 - concepts ☐ Symfony - Avancé
 Enregistrer

Ici, on a joué sur les paramètres multiple et expanded

```
$formulaire = $this->createFormBuilder($seminaire)
    ->add('datedebutsem', DateTime::class, array('label' => 'Date début :',
        'data' => new \DateTime(),))
    ->add('cours', EntityType::class, array('class' => 'AppBundle:Cours',
        'label' => 'Cours :',
        'required' => 'true',
        'multiple' => false,
        'choice_label' => 'libelleCours',
        'expanded'=>true,
    ))
    ->add('Enregistrer', SubmitType::class)
    ->getForm();
```

4. Les thèmes

Un cours a plusieurs thèmes et un thème a plusieurs cours....

On va donc créer une Entity Theme qui a comme attribut id (généré automatiquement) et un attribut libelle en string de 40 caractères.

On va ensuite créer une relation ManyToMany bi-directionnelle entre les 2 Entities.

Côté Cours :

```
/**
 * @ORM\ManyToMany(targetEntity="AppBundle\Entity\Theme", inversedBy="lesCours")
 * @ORM\JoinTable(
 *     name="CoursTheme",
 *     joinColumns={@ORM\JoinColumn(name="idCours", referencedColumnName="id")},
 *     inverseJoinColumns={@ORM\JoinColumn(name="idTheme", referencedColumnName="id")}
 * )
 */
private $lesThemes;
```

Dans l'entity Theme : la relation ManyToMany puisqu'elle est bidirectionnelle :

```
/**
 *
 * @ORM\ManyToMany(targetEntity="AppBundle\Entity\Cours", mappedBy="lesThemes")
 *
 */
private $lesCours;
```

N'oubliez pas de créer les getters, setters et constructeurs avec la commande :
doctrine:generate:entities



Exercice: : Créer le formulaire de saisie d'un thème qui est affiché avec l'url :

http://symfony.br/TPSymfonyDoctrine/web/app_dev.php/creertheme

La saisie du libellé du thème doit être obligatoire.

Libellé du Thème :

Enregistrer

Saisir les thèmes suivants :

id	libelle
1	Développement web
2	Développement mobile
3	Développement système
4	Système et réseau
5	Sécurité informatique

Création des cours avec les thèmes correspondants

On va créer une nouvelle route et une nouvelle action dans le contrôleur :

http://symfony.br/TPSymfonyDoctrine/web/app_dev.php/creerCoursTheme

Code du contrôleur :

```

public function creerCoursThemeAction(Request $request) {
    $cours = new Cours();
    $formulaire = $this->createFormBuilder($cours)
        ->add('libellecours', TextType::class, array('label' => 'Libellé du cours :'))
        ->add('nbjours', IntegerType::class, array('label' => 'Nombre de jours :'))
        ->add('lesThemes', EntityType::class, array('class' => 'AppBundle:Theme',
            'label' => 'libelle',
            'choice_label' => 'libelle',
            'required' => true,
            'multiple' => true,
            'expanded' => true,
        ) )
        ->add('Enregistrer', SubmitType::class)
        ->getForm();
    $formulaire->handleRequest($request);
    if ($formulaire->isValid()) {
        // $cours = $formulaire->getData();
        $em = $this->getDoctrine()->getManager();
        $em->persist($cours);
        $em->flush();
        return $this->render('AppBundle:Appli:ok.html.twig', array('message' => "Cours Créé"));
    }
    return $this->render('AppBundle:Appli:courstheme.html.twig', array('leFormulaire' => $formulaire->createView()));
}

```

Puis créer le fichier coursTheme.html.twig dans le dossier Resources\views\Appli

Le code de ce fichier :



Ne pas oublier les tags `{{ form_start(leFormulaire) }}` et `{{ form_end(leFormulaire) }}` qui délimitent le formulaire. Ils sont l'équivalent des éléments HTML `<form>` et `</form>`

```

<html>
    <head>
        <title>Création d'un cours</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
        {% block leFormulaire %}
            {{ form_start(leFormulaire) }}
            {{ form_row(leFormulaire.libellecours) }}
            {{ form_row(leFormulaire.nbjours) }}
            <div>
                <p>
                    {% for unTheme in leFormulaire.lesThemes %}
                        {{ form_widget(unTheme) }}
                        {{ form_label(unTheme) }}
                        <br>
                    {% endfor %}
                </p>
            </div>
            {{ form_widget(leFormulaire.Enregistrer) }}
            {{ form_end(leFormulaire) }}
        {% endblock %}
    </body>
</html>

```

Ce qui donne au niveau du formulaire :

Saisir 2 cours :

Et voir les modifications dans la bd (tables cours et cours_theme) :

id	libelleCours	nbJours
1	Symfony 3 - concepts	5
2	Symfony - Avancé	4
3	Symfony Doctrine	3
4	Sécurité et android	2
NULL	NULL	NULL

idCours	idTheme
3	1
3	2
4	2
4	4
4	5
NULL	NULL

http://symfony.br/TPSymfonyDoctrine/web/app_dev.php/creerCoursTheme

Exercice :

Afficher un cours et modifier ses thèmes

url : http://symfony.br/TutoSymfonyDoctrine/web/app_dev.php/voircours/4

4 est l'id du cours

idCours	idTheme
3	1
3	2
4	2
4	3
4	4
4	5
	NULL

Partie 3 : Les formulaires réutilisables

Le principe des formulaires va rester le même. L'idée maintenant est de faire un peu mieux.

- ✓ Actuellement : le formulaire est construit dans le contrôleur et donc utilisable uniquement dans cette action du contrôleur.
- ✓ Amélioration : on va bâtir un formulaire réutilisable.



L'idée : créer un constructeur de formulaire.

Ce constructeur est une classe appelée XXXType qui hérite de la classe AbstractType. XXX représente le nom de l'entity associée à ce formulaire.

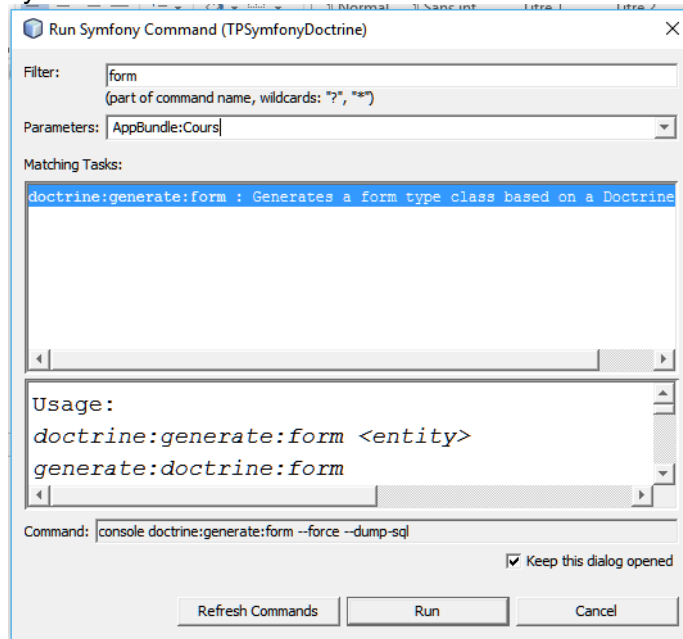
Par exemple : *CoursType*.

Cette classe sera créée dans un nouveau dossier du bundle, le dossier Form

Pour créer ce constructeur de formulaire, il existe la commande Symfony :

Bin\console doctrine:generate:form

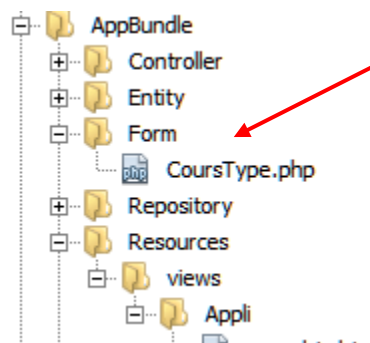
Exemple pour l'entity Cours :



Et voici ce qui a été généré :

```
Symfony 3 (TPSymfonyDoctrine) x | Symfony 3 (TutoSymfonyBR_DoctrineV32) x | Symfony 3 (TPSymfonyDoctrine) #2 x |
"C:\wamp64\bin\php\php7.0.10\php.exe" "T:\Wampsites\CoursSymfony\TPSymfonyDoctrine\bin\console" "--ansi" "doctrine:generate:form" "AppBundle:Cours"
created .\src\AppBundle/Form/
created .\src\AppBundle/Form/CoursType.php
The new CoursType.php class file has been created under T:\Wampsites\CoursSymfony\TPSymfonyDoctrine\src\AppBundle/Form\CoursType.php.
Done.
```

On voit un nouveau dossier :



Voyons le code du fichier CoursType.php

```
class CoursType extends AbstractType
{
    /**
     * {@inheritdoc}
     */
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder->add('libelleCours')->add('nbJours')->add('lesThemes')
    }

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

    /**
     * {@inheritdoc}
     */
    public function getBlockPrefix()
    {
        return 'appbundle_cours';
    }
}
```

Ce code est insuffisant, il faut reprendre la méthode buildForm pour spécifier les types de champs, ce qui est plus prudent si on veut éviter les erreurs.

Le code du contrôleur :

```

/**
 *
 * On va recréer un cours pour permettre la saisie des thèmes qui vont bien
 * @Route("/voirCoursForm/{id}")
 * name="voirCoursFormId"
 * requirements={"id":"\d+"}
 *
 */
public function voirCoursFormAction(Request $request, $id) {
    $em = $this->getDoctrine()->getManager();
    $repository = $em->getRepository('AppBundle:Cours');
    $cours = $repository->find($id);
    $formulaire = $this->get('form.factory')->create(\AppBundle\Form\CoursType::class, $cours);
    $formulaire->handleRequest($request);
    //$this->get('request');
    if ($formulaire->isValid()) {
        //$cours=$formulaire->getData();
        $em->persist($cours);
        $em->flush();
        return $this->render('AppBundle:Appli:ok.html.twig', array('message' => "Cours avec thèmes créé"));
    }
    return $this->render('AppBundle:Appli:courstheme.html.twig', array('leFormulaire' => $formulaire->createView()));
}

```



On aurait pu créer un formulaire réutilisable pour les thèmes, mais il aurait fallu gérer la relation ManyToMany entre les cours et les thèmes.

Testez :

http://symfony.br/TutoSymfonyBR_Doctrine/web/app_dev.php/voircoursform/4

```

public function buildForm(FormBuilderInterface $builder, array $options)
{
    $builder
        ->add('libellecours', 'text', array('label' => 'Libellé du cours :'))
        ->add('nbjours', 'integer', array('label' => 'Nombre de jours :'))
        ->add('themes', 'entity', array(
            'class' => 'AppBundle:Theme',
            'property' => 'libelle',
            'label' => 'Thèmes :',
            'required' => true,
            'multiple' => true,
            'expanded' => true
        ))
        ->add('Enregistrer', 'submit')
}

```

5. Les formulaires réutilisables



Un conseil
Voir la commande

doctrine:generate:crud

... elle devrait aussi vous rendre des services !!!

Refaites ces exercices en partant de CRUD et en améliorant les interfaces, notamment en utilisant bootstrap.



Bon courage !!!