



## 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\_Doctrine

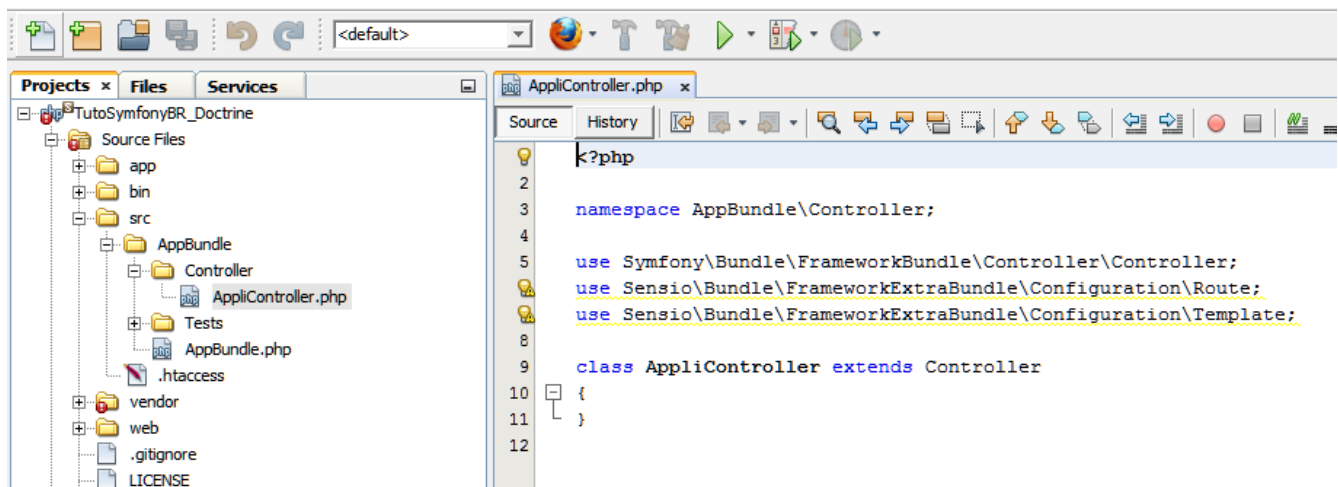
✓ Vous utiliserez le bundle AppBundle

Faire un test avec l'URL :

[http://symfony.br/TutoSymfonyBR\\_Doctrine/web/app\\_dev.php](http://symfony.br/TutoSymfonyBR_Doctrine/web/app_dev.php)

✓ Supprimer le contrôleur DefaultController

✓ Créer le contrôleur AppliController (classe AppliController, fichier AppliController.php). Pas d'action pour l'instant



## Partie 1 : la structure

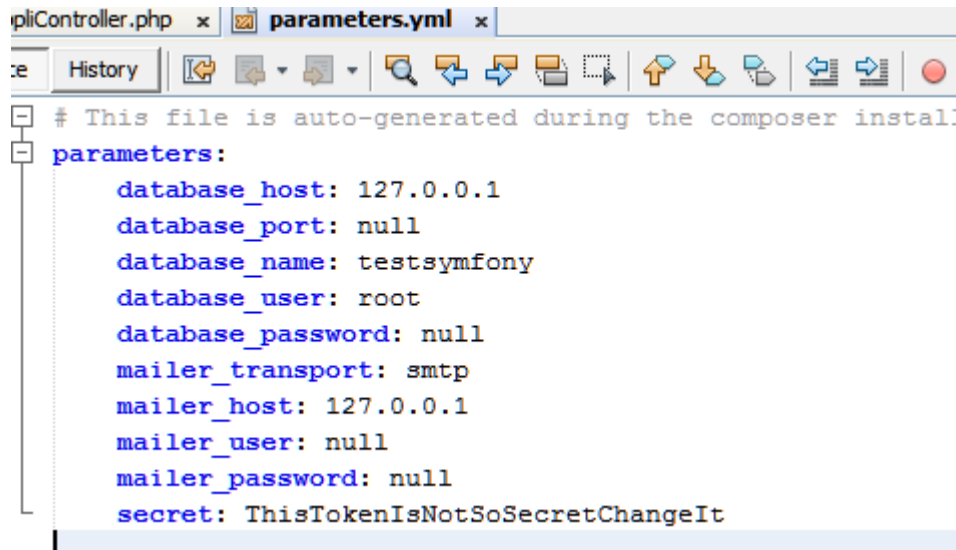
On va commencer par saisir les paramètres de la base de données. Pour ceci, allez dans le fichier parameters.yml :

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: testsymfony

- ✓ database\_user: root
- ✓ database\_password: null



```

# This file is auto-generated during the composer instal.
parameters:
    database_host: 127.0.0.1
    database_port: null
    database_name: testsymfony
    database_user: root
    database_password: null
    mailer_transport: smtp
    mailer_host: 127.0.0.1
    mailer_user: null
    mailer_password: null
    secret: ThisTokenIsNotSoSecretChangeIt

```



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%"

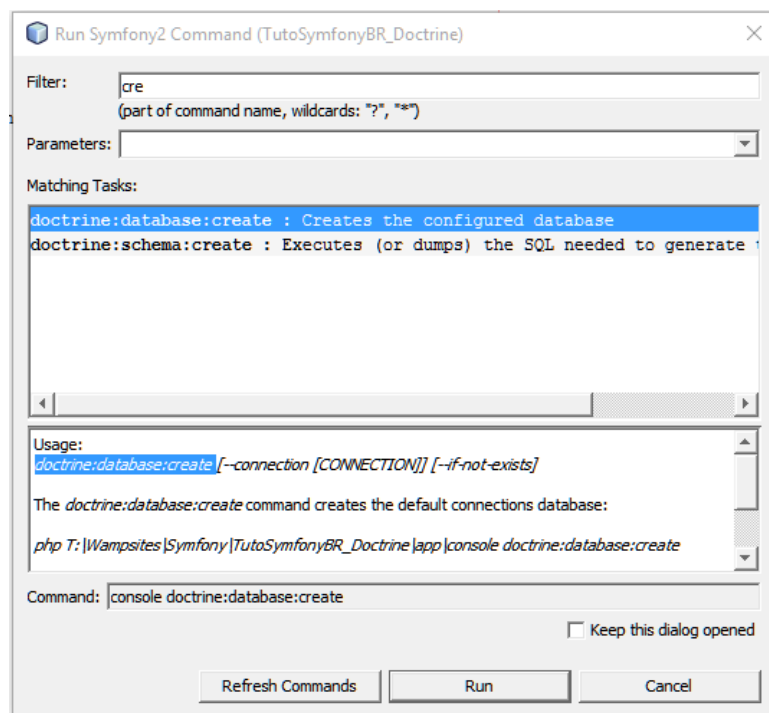
    orm:

```

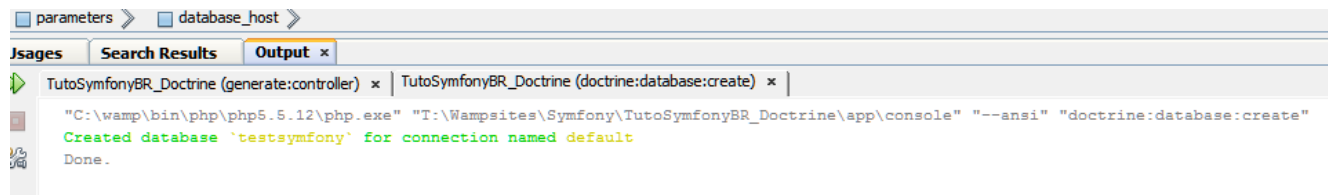
On remarque que le driver est indiqué en dur ... pdo\_mysql. Les autres paramètres seront mappés sur les paramètres correspondants du fichier config.yml.

Par exemple host: "%database\_host%" récupérera la valeur du paramètre database\_host: 127.0.0.1 du fichier parameters.yml

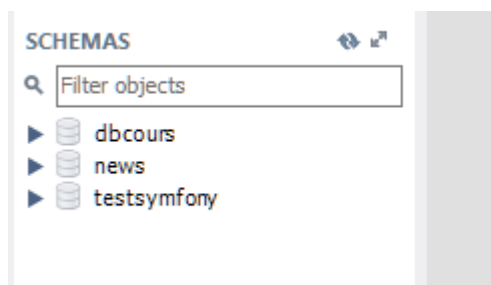
Vous allez créer la base de données testsymfony à l'aide de la commande symfony2 doctrine:database:create



La base de données va donc se créer.

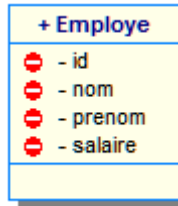


On va le vérifier avec le client mysql workbench :



La base a bien été créée.

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

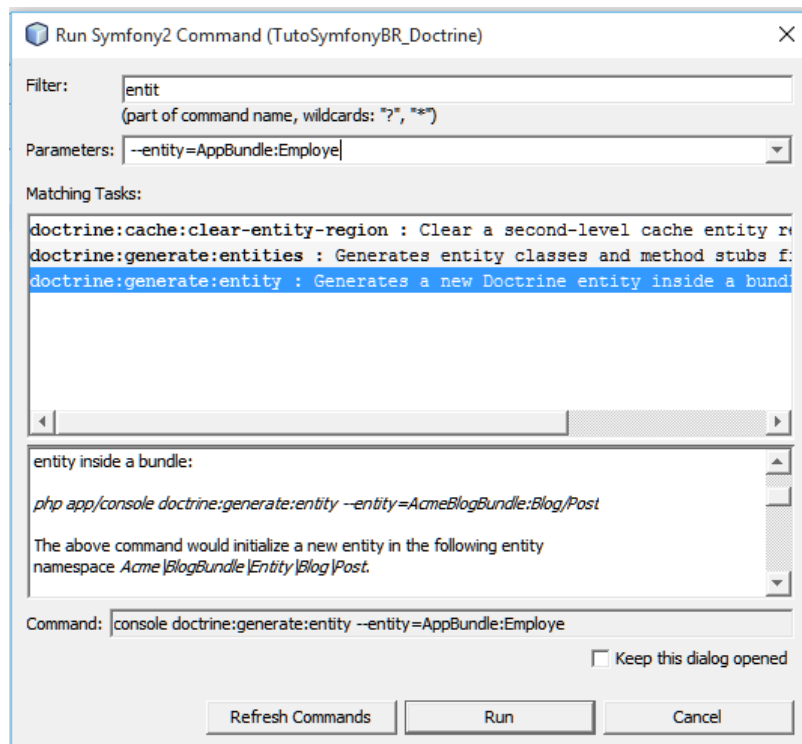


id est un entier, nom et prénom des chaînes de caractères et salaire un décimal

Pour faire ceci, ouvrez une console de commande symfony2 et appelez la commande `generate:doctrine:entities`

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

TutoSymfonyBR\_Doctrine (generate:controller) x | TutoSymfonyBR\_Doctrine (doctrine:generate:entity)

```
"C:\wamp\bin\php\php5.5.12\php.exe" "T:\Wampsites\Symfony\TutoSymfonyBR_Doctrine"
```

```
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]:
```

Ensuite 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

```
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.
```

```
New field name (press <return> to stop adding fields): 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 265. Saisir 50.

```
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)

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

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

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): prenom
Field type [string]:
Field length [255]: 50

New field name (press <return> to stop adding fields): salaire
Field type [string]: decimal

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

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.

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

```
Summary before generation
```

```
You are going to generate a "AppBundle:Employee" Doctrine2 entity  
using the "annotation" format.
```

```
Do you confirm generation [yes]? |
```

Il ne vous reste plus qu'à confirmer la création de l'entity et de son repository en faisant enter.

La génération a bien eu lieu :

```
Do you confirm generation [yes]?
```

```
Entity generation
```

```
Generating the entity code: OK
```

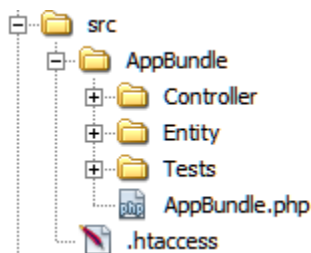
```
You can now start using the generated code!
```

```
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 un nouveau dossier Entity dans le dossier de notre bundle



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

- ✓ Employee.php pour la classe Employee
- ✓ EmployeeRepository pour son Repository.

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

k?php

```
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * Employe
 *
 * @ORM\Table()
 * @ORM\Entity(repositoryClass="AppBundle\Entity\EmployeeRepository")
 */
class Employee
{
    /**
     * @var integer
     *
     * @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;

    /**
     * @var string
     */
}
```



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 :

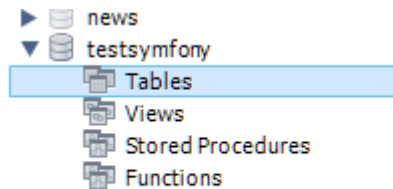
```
k?php

namespace AppBundle\Entity;

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

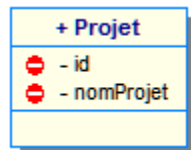
On la complètera ultérieurement.

Voyons côté BD :



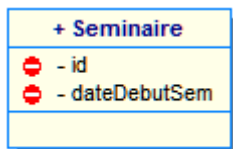
La base testsymfony ne contient encore aucune table. C'est normal, on n'a pas demandé à Doctrine de mettre à k=jour ce schéma (cette base).

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



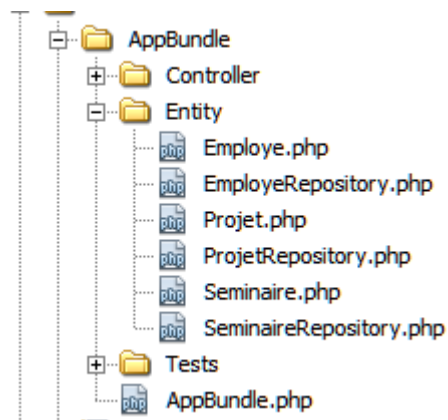
id : sera généré automatiquement  
nomProjet : chaîne de 50 caractères

et



id : sera généré automatiquement  
dateDebutSem format date

Vous devriez maintenant avoir les classes suivantes dans le dossier Entity de votre bundle AppBundle :



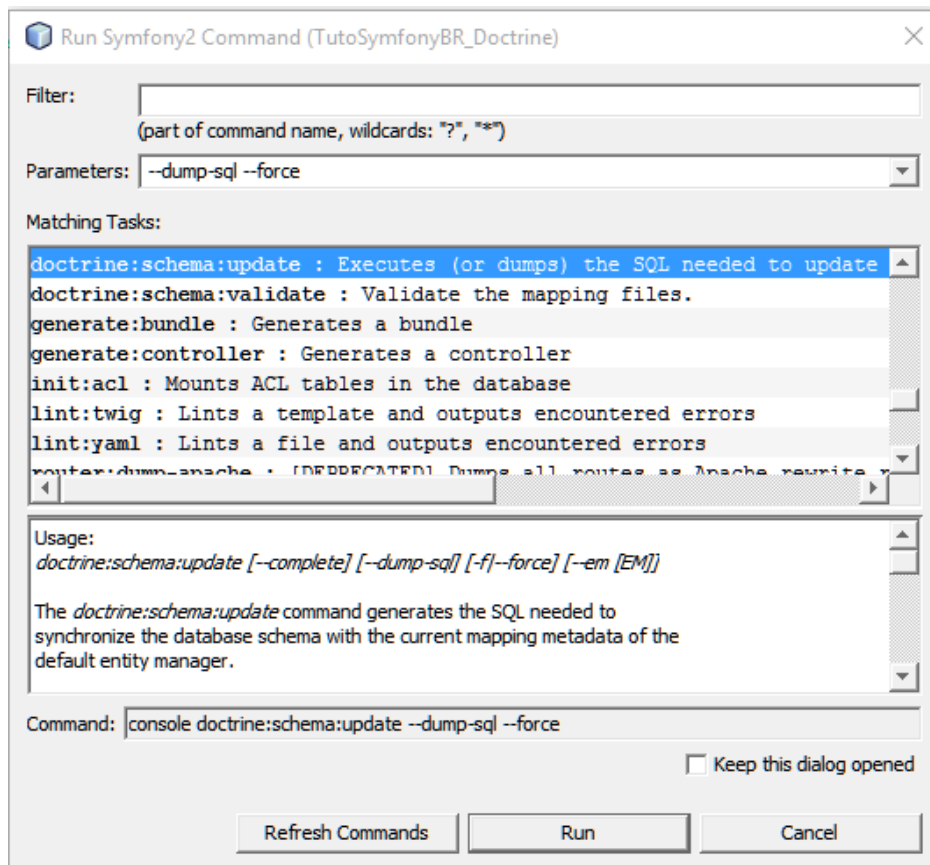
Je vous conseille d'ouvrir les fichiers Entity pour vérifier les informations contenues. Sont-elles conformes à ce qui est demandé ?

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

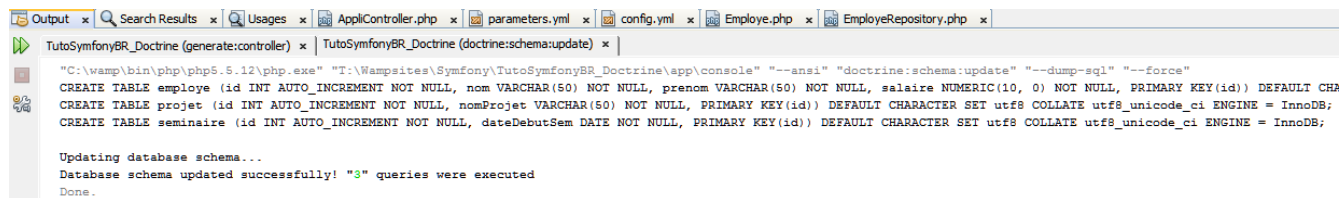
Avec les bonnes options ...

--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 ...



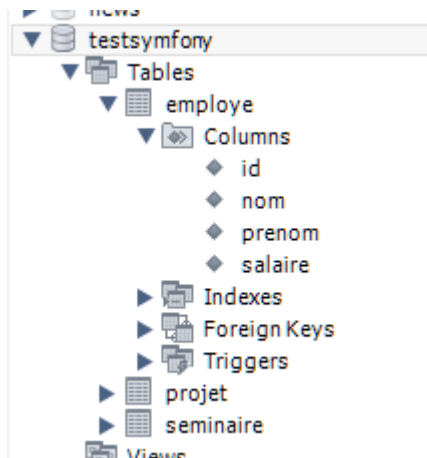
Dans la fenêtre de résultat on voit bien les ordres SQL générés :



```
"C:\wamp\bin\php\php5.5.12\php.exe" "T:\Wampsites\Symfony\TutoSymfonyBR_Doctrine\app\console" "--ansi" "doctrine:schema:update" "--dump-sql" "--force"
CREATE TABLE employe (id INT AUTO_INCREMENT NOT NULL, nom VARCHAR(50) NOT NULL, prenom VARCHAR(50) NOT NULL, salaire NUMERIC(10, 0) 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;
```

```

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

/**
 * Set salaire
 *
 * @param string $ville
 *
 * @return Employe
 */
public
    function setVille($ville) {
        $this->ville = $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 :

```

Output x Search Results x Usages x AppliController.php x parameters.yml x config.yml x Employee.php x EmployeeReposit
TutoSymfonyBR_Doctrine (generate:controller) x TutoSymfonyBR_Doctrine (doctrine:schema:update) x
"C:\wamp\bin\php\php5.5.12\php.exe" "T:\Wampsites\Symfony\TutoSymfonyBR_Doctrine\app\console" "--ansi" "doctrine:sche
ALTER TABLE employe ADD ville VARCHAR(40) DEFAULT 'Toulon' NOT NULL;

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

```

Seules les modifications ont été prises en compte !!!

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("Employe", indexes={@ORM\Index(name="ind_nom", columns={"nom"}),
 * @ORM\Index(name="ind_ville", columns={"ville"})})
 *
 * @ORM\Entity(repositoryClass="AppBundle\Entity\EmployeeRepository")

```

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

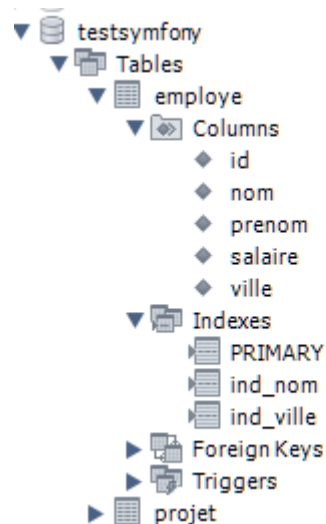
```

Output x
TutoSymfonyBR_Doctrine (generate:controller) x | TutoSymfonyBR_Doctrine (doctrine:schema:update) x |
"C:\wamp\bin\php\php5.5.12\php.exe" "T:\Wampsites\Symfony\TutoSymfonyBR_Doctrine\ap
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 !!!!

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 :

```
/**  
 * 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 !!!

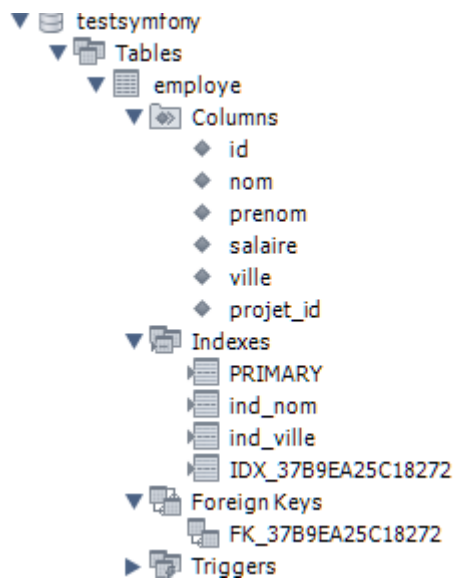
```
ty\Projet $projet = null) {
```

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

```
TutoSymfonyBR_Doctrine (generate:controller) x | TutoSymfonyBR_Doctrine (doctrine:schema:update) x |
"C:\wamp\bin\php\php5.5.12\php.exe" "T:\Wampsites\Symfony\TutoSymfonyBR_Doctrine\app\console" "--ansi" "d
ALTER TABLE employe ADD projet_id INT DEFAULT NULL;
ALTER TABLE employe ADD CONSTRAINT FK_37B9EA25C18272 FOREIGN KEY (projet_id) REFERENCES projet (id);
CREATE INDEX IDX_37B9EA25C18272 ON employe (projet_id);

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

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.



***on aurait pu peaufiner nos annotations pour ne pas avoir ces noms d'index et de contrainte barbares !!!***

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 annotations ....en soignant les noms à attribuer !!!

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

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

Et l'annotation suivante dans l'entity Seminaire :

```
/**
 *
 * @ORM\ManyToMany(targetEntity="\AppBundle\Entity\Employe", inversedBy="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 : employe\_id et seminaire\_id,
- ✓ L'attribut mappedBy 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 inversedBy qui est l'attribut inverse de l'entity Employe.

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

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

Et le résultat côté symfony :

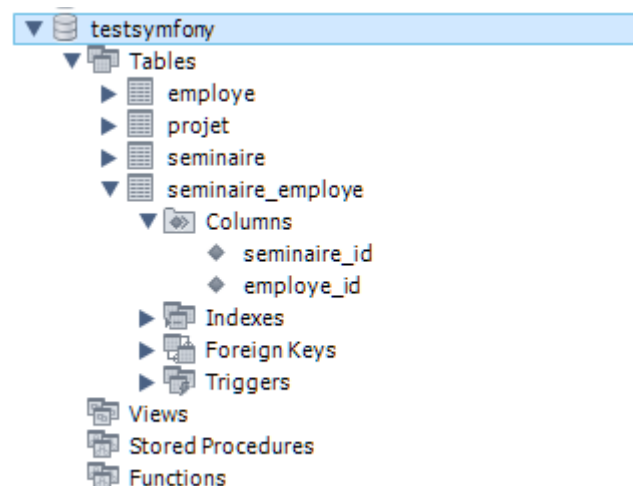
```

Output x
TutoSymfonyBR_Doctrine (generate:controller) x | TutoSymfonyBR_Doctrine (doctrine:schema:update) x |
"C:\wamp\bin\php\php5.5.12\php.exe" "T:\Wampsites\Symfony\TutoSymfonyBR_Doctrine\app\console" "--ansi" "doctrine:schema:update" "--dump-sql" "--force"
CREATE TABLE seminaire_employe (seminaire_id INT NOT NULL, employe_id INT NOT NULL, INDEX IDX_403F518ACEA14D8 (seminaire_id), INDEX IDX_403F518A1B65292 (employe_id), PRIMARY KEY(seminaire_id, employe_id))
ALTER TABLE seminaire_employe ADD CONSTRAINT FK_403F518ACEA14D8 FOREIGN KEY (seminaire_id) REFERENCES seminaire (id) ON DELETE CASCADE;
ALTER TABLE seminaire_employe ADD CONSTRAINT FK_403F518A1B65292 FOREIGN KEY (employe_id) REFERENCES Employe (id) ON DELETE CASCADE;

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

```

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



La table seminaire\_employe 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 symfony2, vous allez créer l'entity cours et une relation ManyToOne entre les entites Seminaire et Cours

Entity Cours :

Id (généré automatiquement), libellecours (string(40)), nbjours (integer)

Dans Seminaire :

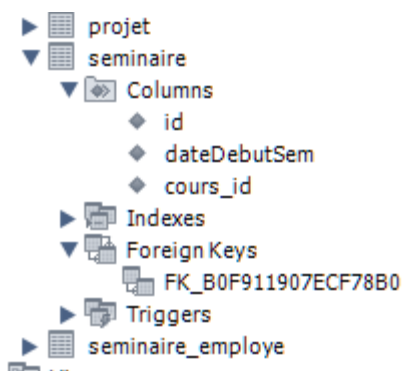
```
/**
 *
 * @ORM\ManyToOne(targetEntity="\AppBundle\Entity\Cours")
 */
private $cours;
```

Ne pas oublier de rajouter l'accesseur et le mutateur pour le cours :

```
public function getCours() {
    return $this->cours;
}

public function setCours($cours) {
    $this->cours=$cours;
}
```

On devrait avoir ceci dans la BD :



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 (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 ...

```
<?php

namespace AppBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Template;
use Symfony\Component\HttpFoundation\Request;
use AppBundle\Entity\Cours;

class AppliController extends Controller {

    /**
     * @Route("/creercours")
     */
    public function creerCoursAction(Request $request) {
        $cours = new Cours();
        $formulaire = $this->createFormBuilder($cours)
            ->add('libellecours', 'text', array('label' => 'Libellé du cours :'))
            ->add('nbjours', 'integer', array('label' => 'Nombre de jours :'))
            ->add('Enregistrer', 'submit')
            ->getForm();
        $formulaire->handleRequest($request);
        $this->get('request');
        if ($formulaire->isValid()) {
            $em = $this->getDoctrine()->getManager();
            $em->persist($cours);
            $em->flush();
            return $this->render('AppBundle:App:ok.html.twig', array('message' => "Cours Créé"));
        }
        return $this->render('AppBundle:App:cours.html.twig', array('leFormulaire' => $formulaire->createView()));
    }
}
```

La route : /creercours

### **La nouveauté :**

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

```

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

```



Et voici le formulaire :

[http://symfony.br/TutoSymfonyBR\\_Doctrine/web/app\\_dev.php/creercours](http://symfony.br/TutoSymfonyBR_Doctrine/web/app_dev.php/creercours)

Que l'on va remplir :

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 :

Vérifions :

Result Grid     Filter Rows: <input type="text"/>   Ex			
	id	libellecours	nbjours
▶	2	Symfony2 Concepts	5
*	NULL	NULL	NULL

L'objet a bien été persisté en enregistrement.

### 3. Le formulaire Séminaire

---

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

```

bdlb_appli_creerseminaire:
    pattern: /creerseminaire
    defaults: { _controller: BdlbAppliBundle:Appli:creerSeminaire }

```

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 ... seminaire ?

```

/**
 * @Route("/creerseminaire")
 *
 */
public function creerSeminareAction(Request $request) {
    $seminaire = new Seminaire();
    $formulaire = $this->createFormBuilder($seminaire)
        ->add('datedebutsem', 'date', array('label' => 'Date début :'))
        ->add('cours', 'entity', array('class' => 'AppBundle:Cours',
            'property' => 'libellecours',
            'label' => 'Cours :',
            'required' => 'true',
            'empty_value' => 'Choisir un cours'))
        ->add('Enregistrer', 'submit')
        ->getForm();
    $formulaire->handleRequest($request);
    $this->get('request');
    if ($formulaire->isValid()) {
        $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:App: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 et qu'à l'affichage du formulaire on ait une invitation à choisir le cours. La liste déroulante sera automatiquement remplie avec les valeurs des attributs libellecours des cours de l'Entity Cours : [http://symfony.br/TutoSymfonyBR\\_Doctrine/web/app\\_dev.php/creerseminaire](http://symfony.br/TutoSymfonyBR_Doctrine/web/app_dev.php/creerseminaire)

TODO supply a title

← symfony.br/TutoSymfonyBR\_Doctrine/web/app\_dev.php/creersemini

Date début :

Jan 1 2010

Cours : Choisir un cours

Enregistrer

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

TODO supply a title

← symfony.br/TutoSymfonyBR\_Doctrine/web/app\_dev.php/creersemini

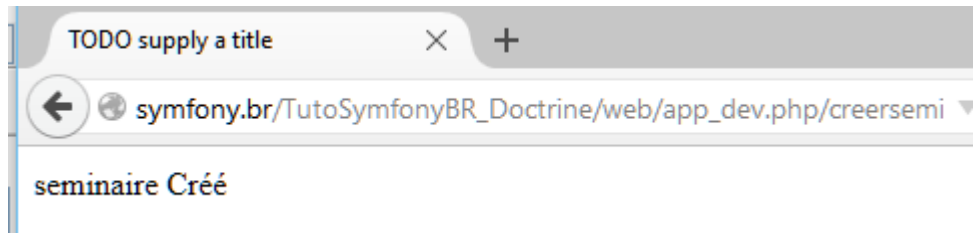
Date début :

Nov 17 2015

Cours : Symfony2 Concepts

Enregistrer

A l'écran :



Dans la base de données :

	id	dateDebutSem	cours_id
▶	1	2015-11-17	2
★	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

Autre représentation possible :

A screenshot of a web form in a browser. The tab is 'TODO supply a title'. The URL is 'symfony.br/TutoSymfonyBR\_Doctrine/web/app\_dev.php/creerseminairecours'. The form has a 'Date début :' section with dropdowns for 'Jan', '1', and '2010'. Below that is a 'cours :' section with two radio buttons: 'Symfony2 Concepts' and 'Symfony Avancé'. At the bottom is an 'Enregistrer' button.

Ici, on a joué sur les paramètres multiple et expanded

```

/**
 * @Route("/creerseminairecours")
 *
 */
public function creerSeminaireCoursAction(Request $request) {
    $seminaire = new Seminaire();
    $formulaire = $this->createFormBuilder($seminaire)
        ->add('datedebutsem', 'date', array('label' => 'Date début :'))
        ->add('Cours', 'entity', array('class' => 'AppBundle:Cours',
            'label' => 'cours :',
            'required' => true,
            'multiple' => false,
            'expanded' => true))
        ->add('Enregistrer', 'submit')
        ->getForm();
    $formulaire->handleRequest($request);
    $this->get('request');
    if ($formulaire->isValid()) {
        $sem = $this->getDoctrine()->getManager();
        $sem->persist($seminaire);
        $sem->flush();
        return $this->render('AppBundle:App:ok.html.twig', array('message' => "seminaire Créé"));
    }
    return $this->render('AppBundle:App:cours.html.twig', array('leFormulaire' => $formulaire->createView()));
}

```

On n'a pas non plus indiqué le paramètre property car on a créé la méthode suivante dans la classe Cours :

```

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

```

#### 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 (généralisé 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="Theme" ,
 *                 inversedBy="cours",
 *                 cascade={"persist"})
 *
 */
private $themes;

```



Sans oublier :

- ✓ D'instancier la collection \$themes dans le constructeur que l'on va créer à cet effet :

```
public function __construct() {  
    $this->themes = new \Doctrine\Common\Collections\ArrayCollection();  
}
```

- ✓ De rajouter le getter et le setter :

```
public function getThemes() {  
    return $this->themes;  
}  
  
public function setThemes(\Doctrine\Common\Collections\ArrayCollection $themes)  
{  
    $this->themes = $themes;  
}
```

Enfin, dans l'entity Theme :

La relation ManyToMany puisqu'elle est bidirectionnelle :

```
/**  
 * @ORM\ManyToMany(targetEntity="Cours" , mappedBy="themes")  
 */  
private $courss;
```

On note \$courss : **collection d'objets de la classe Cours**

Le constructeur :

```
public function __construct() {  
    $this->courss = new \Doctrine\Common\Collections\ArrayCollection();  
}
```

Le getter et le setter :

```
public function getCourss() {  
    return $this->courss;  
}  
  
public function setCourss(\Doctrine\Common\Collections\ArrayCollection $courss) {  
    $this->courss = $courss;  
}
```



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

[http://symfony.br/TutoSymfonyBR\\_Doctrine/web/app\\_dev.php/creertheme](http://symfony.br/TutoSymfonyBR_Doctrine/web/app_dev.php/creertheme)

La saisie du libellé du thème doit être obligatoire.

Libellé du thème :

Saisir les thèmes suivants :

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

Enfin, on modifie le formulaire pour la classe (Entity) Cours

En fait on va créer une nouvelle route et une nouvelle action dans le contrôleur (uniquement pour les besoins du cours)

Url :

[symfony.br/TutoSymfonyBR\\_Doctrine/web/app\\_dev.php/creerCoursTheme](http://symfony.br/TutoSymfonyBR_Doctrine/web/app_dev.php/creerCoursTheme)

Code du contrôleur :

```

public function creerCoursThemeAction(Request $request) {
    $cours = new Cours();
    $formulaire = $this->createFormBuilder($cours)
        ->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',
            'empty_value' => 'Choisir un cours',
            'multiple' => true,
            'expanded' => true
        ))
        ->add('Enregistrer', 'submit')
        ->getForm();
    $formulaire->handleRequest($request);
    $this->get('request');
    if ($formulaire->isValid()) {
        // $cours = $formulaire->getData();
        $em = $this->getDoctrine()->getManager();
        $em->persist($cours);
        $em->flush();
        return $this->render('AppBundle:App:ok.html.twig', array('message' => "Cours avec thèmes créé"));
    }
    return $this->render('AppBundle:App:coursthemes.html.twig', array('leFormulaire' => $formulaire->createView()));
}

```

Ce qui donne au niveau du formulaire :

TODO supply a title

symfony.br/TutoSymfonyBR\_Doctrine/web/app\_dev.php/creerCoursTheme

Libellé du cours :

Nombre de jours :

☐ Développement web

☐ Développement mobile

☐ Développement système

☐ Système et réseau

☐ Sécurité informatique

Enregistrer

On a au préalable créé un autre formulaire pour un affichage plus lisible :

```

<body>
    {% block leFormulaire %}
        {{ form_start(leFormulaire) }}
        {{ form_row(leFormulaire.libellecours) }}
        {{ form_row(leFormulaire.nbjours) }}
        <div>
            <p>
                {% for unTheme in leFormulaire.themes %}
                    {{ form_widget(unTheme) }}
                    {{ form_label(unTheme) }}
                    <br>
                {% endfor %}
            </p>
        </div>
        {{ form_widget(leFormulaire.Enregistrer) }}
        {{ form_end(leFormulaire) }}
    {% endblock %}

</body>

```



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>`

Saisir 2 cours :

← symfony.br/TutoSymfonyBR\_Doctrine/web/app\_dev.php/creerCoursTheme

Libellé du cours :

Nombre de jours :

☒ Développement web  
☒ Développement mobile  
☐ Développement système  
☐ Système et réseau  
☐ Sécurité informatique

Et

TODO supply a title × +

← → symfony.br/TutoSymfonyBR\_Doctrine/web/app\_dev.php/creerCoursTheme

Libellé du cours :

Nombre de jours :

☐ Développement web  
☒ Développement mobile  
☐ Développement système  
☒ Système et réseau  
☒ Sécurité informatique

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

	id	libellecours	nbjours
▶	2	Symfony2 Concepts	5
	3	Symfony Avancé	3
	4	Symfony Doctrine	3
	5	Sécurité et android	3

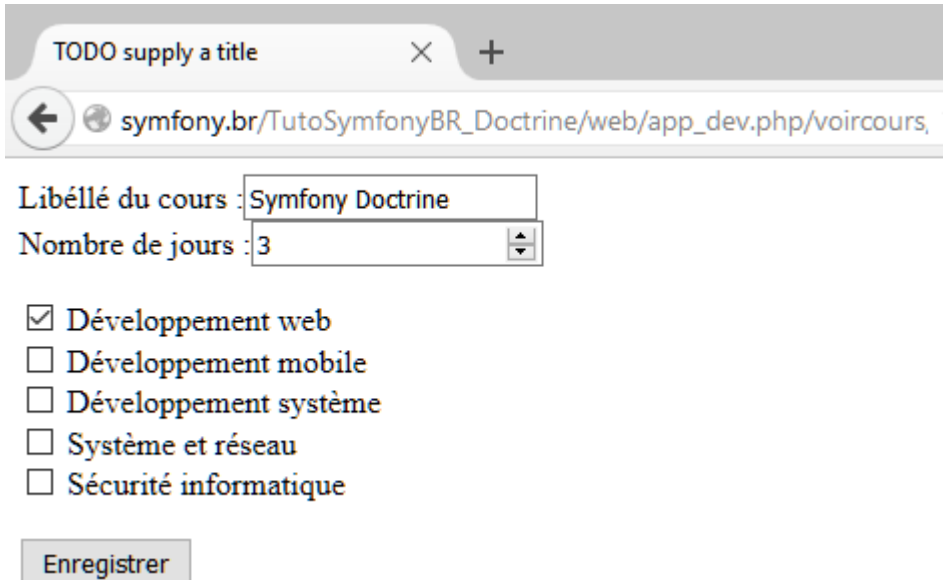
	cours_id	theme_id
	4	1
	4	2
	5	2
	5	4
	5	5
	NULL	NULL

Exercice :

Afficher un cours et modifier ses thèmes

url : [http://symfony.br/TutoSymfonyBR\\_Doctrine/web/app\\_dev.php/voircours/4](http://symfony.br/TutoSymfonyBR_Doctrine/web/app_dev.php/voircours/4)

4 est l'id du cours



The screenshot shows a web browser window with a single tab titled "TODO supply a title". The address bar displays the URL "symfony.br/TutoSymfonyBR\_Doctrine/web/app\_dev.php/voircours/". Below the browser, a form is visible with the following elements:

- A label "Libellé du cours :" followed by a text input field containing "Symfony Doctrine".
- A label "Nombre de jours :" followed by a spinner input field containing the value "3".
- A list of checkboxes for course themes:
  - ☒ Développement web
  - ☐ Développement mobile
  - ☐ Développement système
  - ☐ Système et réseau
  - ☐ Sécurité informatique
- A button labeled "Enregistrer" at the bottom.

### 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.

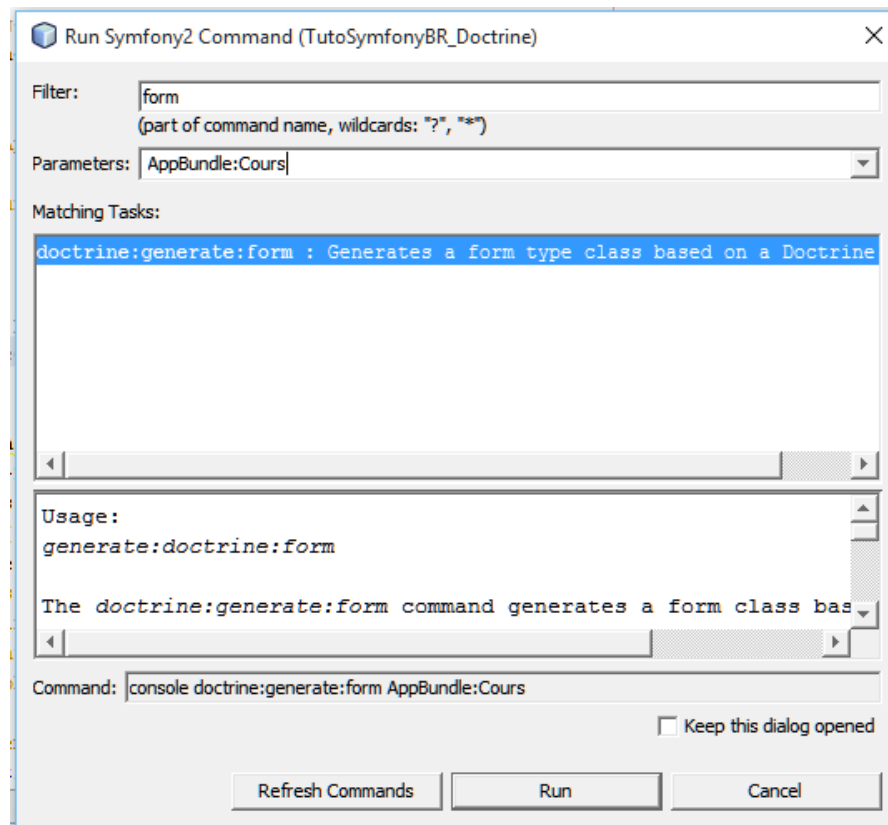
L'idée : créer un constructeur de formulaire. Ce constructeur est une classe appelée XXXType qui hérite de la classe AbstractType. XXX représente le nom de l'entity associée à ce formulaire. Par exemple CoursType.

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

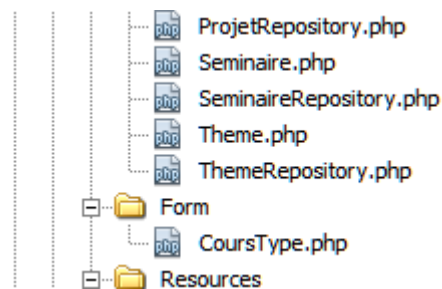
Pour créer ce constructeur de formulaire, il existe la commande symfony2 :

```
console doctrine:generate:form
```

Exemple pour l'entity Cours :



Et voici ce qui a été généré :



Et le code :

⌘?php

```
namespace AppBundle\Form;

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

class CoursType extends AbstractType
{
    /**
     * @param FormBuilderInterface $builder
     * @param array $options
     */
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('libellecours')
            ->add('nbjours')
            ->add('themes')
        ;
    }

    /**
     * @param OptionsResolverInterface $resolver
     */
    public function setDefaultOptions(OptionsResolverInterface $resolver)
    {
    }
```

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

Testez :

[http://symfony.br/TutoSymfonyBR\\_Doctrine/web/app\\_dev.php/voircoursform/4](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 !!!



***Bon courage !!!***