

Créer sa première application web en PHP avec Symfony2

Par [Jérôme Place](#)

Date de publication : 14 mai 2011

Dernière mise à jour : 8 août 2011

Ce tutoriel a pour but de vous faire découvrir le framework Symfony2 à travers la création d'une petite application web.

Commentez

I - Prérequis.....	3
II - Objectif du tutoriel.....	3
III - Installation de Symfony.....	3
IV - Création d'un premier bundle avec le générateur.....	4
IV-A - La notion de bundle.....	4
IV-B - Le générateur de bundle.....	4
IV-C - Lien entre le bundle et Symfony2.....	5
IV-D - Afficher un premier message.....	6
V - Comprendre la structure des bundles.....	6
VI - Création des entités.....	7
VII - Création de la base de données et des tables.....	10
VII-A - Configurons notre base de données.....	10
VII-B - Création de la base.....	10
VII-C - Création des tables.....	11
VII-D - Enregistrement d'une première donnée.....	11
VIII - Les templates Twig.....	12
VIII-A - Création d'un premier template.....	12
VIII-B - En savoir plus sur Twig.....	13
IX - Symfony2 et les contrôleurs.....	13
X - Symfony2 et les formulaires.....	15
X-A - Création d'un premier formulaire.....	15
X-B - Le formulaire de modification.....	17
XI - Création d'une page web simple.....	18
XII - Supprimer des données.....	19
XIII - À vous de jouer !.....	19
XIV - Références.....	19
XV - Remerciement.....	20

I - Prérequis

Connaissances requises :

- PHP (niveau intermédiaire à avancé) ;
- base de données (niveau débutant) ;
- (X)HTML (niveau intermédiaire) ;
- programmation orientée objet (niveau débutant).

Il n'est pas nécessaire de connaître les versions antérieures de Symfony (1.2, 1.3 ou 1.4).

Configuration minimale :

- PHP 5.3.2 ;
- serveur http : Apache, IIS, etc. ;
- système de base de données : MySQL, PostgreSQL, SQLite, etc.

II - Objectif du tutoriel

Ce tutoriel a pour objectif de vous apprendre les bases pour développer une application web grâce au langage PHP et au framework Symfony2. Nous allons prendre comme exemple un logiciel de gestion de films :


- chaque film comporte un ou plusieurs acteurs ;
- chaque film est classé dans une catégorie (Comédie, Science-fiction, etc.).

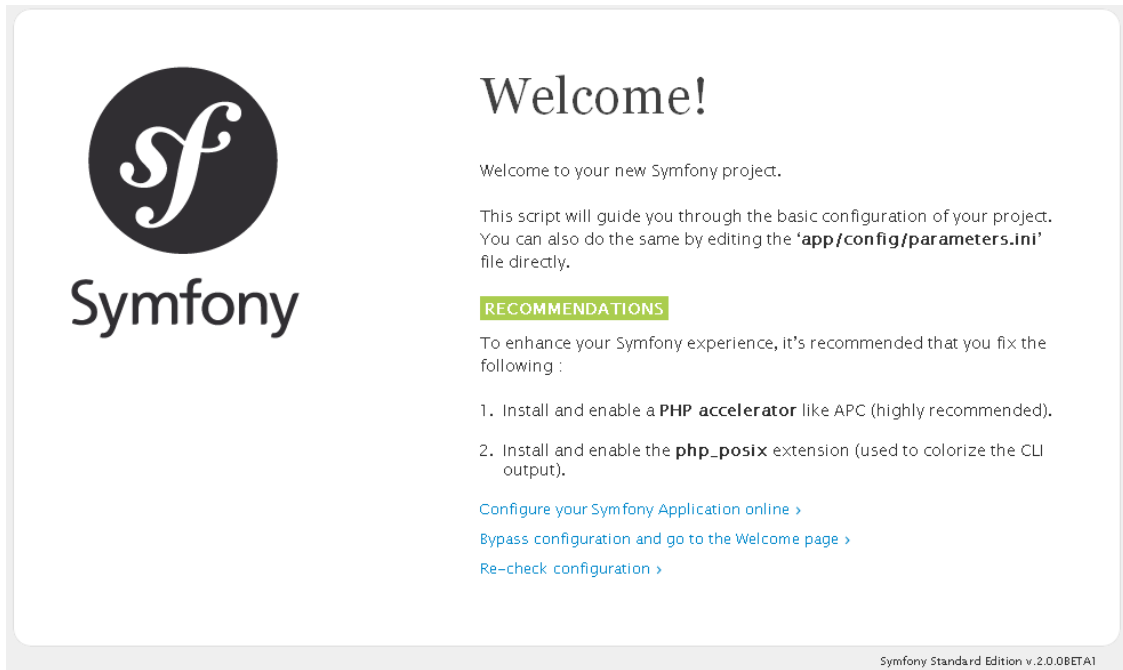
Nous allons ainsi créer une application qui permettra de gérer les acteurs, les catégories et les films.

Cet exemple très simple permettra d'aborder les principaux points pour développer une application web. Nous verrons comment créer les formulaires d'ajout et de modification, les listes, la mise en page, etc.

Nous passerons assez vite sur plusieurs notions de Symfony2 et ne proposons souvent qu'une seule approche même s'il peut exister plusieurs façons d'arriver au même résultat. Cela est volontaire, le but étant d'aller au plus simple avec un exemple concret et non de réaliser une liste exhaustive des fonctionnalités de Symfony2.

III - Installation de Symfony

- 1 Allez sur le site Symfony.com ( www.symfony.com) dans la rubrique "Get Started > Download Symfony" pour télécharger la dernière version de Symfony. À noter que ce tutoriel est basé sur la version finale, Symfony Standard 2.0.0.
- 2 Décompressez le contenu du fichier téléchargé dans le répertoire racine de votre serveur web.
- 3 Renommez le répertoire pour "Symfony2". C'est le nom que nous allons utiliser tout au long du tutoriel.
- 4 Après avoir lancé votre serveur web, vérifiez que Symfony fonctionne correctement en tapant l'adresse "<http://localhost/Symfony2/web/config.php>" dans votre navigateur web préféré (si nécessaire modifiez localhost par l'URL d'accès à votre serveur).
- 5 Si tout va bien, vous devriez alors voir apparaître le message "Welcome" sur la page suivante :



Problèmes possibles :

- rien ne s'affiche : vérifiez la configuration de votre serveur web ;
- Symfony2 vous demande d'installer des extensions PHP : activez ces extensions sans quoi ce framework ne fonctionnera pas correctement. Si besoin consultez la documentation associée à votre serveur web ;
- vous n'y arrivez toujours pas : laissez votre message sur le forum d'aide...

Une fois Symfony2 installé, vous pouvez lancer l'application livrée par défaut en vous rendant sur la page : http://localhost/Symfony2/web/app_dev.php/demo/hello/World. Le message "Hello World" devrait s'afficher.

Symfony2 est installé, voyons maintenant comment créer la structure de base de notre application.

IV - Création d'un premier bundle avec le générateur

IV-A - La notion de bundle

Dans l'univers de Symfony, un morceau d'application est appelé **Bundle**. En fait, cela correspond à une brique logicielle, c'est-à-dire un ensemble cohérent pour une fonctionnalité donnée. Par exemple, pour un site internet, on peut imaginer plusieurs bundles : un bundle actualité, un bundle forum, un bundle newsletter, un bundle utilisateurs, etc. Chaque bundle peut être indépendant, ou bien fonctionner avec d'autres bundles (comme par exemple le forum et ses utilisateurs qui vont poster des messages). Notre application étant très simple, nous n'aurons qu'un seul bundle appelé **Filmotheque**.

IV-B - Le générateur de bundle

Nous pourrions très bien ajouter un dossier **Filmotheque** et commencer à créer un à un les fichiers PHP dont nous avons besoin. Mais l'un des objectifs des frameworks est de gagner du temps. Du coup, les auteurs de Symfony ont mis au point un "générateur de bundle" qui permet de créer automatiquement plusieurs dossiers et fichiers par défaut.

- Ouvrez votre terminal (utilisateurs de Linux) ou bien l'invite de commande (cmd.exe pour les utilisateurs de Windows).
- Allez dans le répertoire Symfony2 en utilisant la commande **cd**.
- Tapez **cd /var/www/Symfony2** (modifiez l'adresse si nécessaire. Par exemple sous Windows : **cd C:\wamp\www\Symfony2**).

- 4 Appuyez sur la touche ENTER.
- 5 Taper la commande : **php app/console**.
- 6 Si la commande **php** n'est pas reconnue, il vous faut installer PHP (utilisateurs Linux) ou l'ajouter dans la variable d'environnement Path (utilisateurs Windows). Voir procédure au paragraphe 3-PATH de la page : <http://trac.symfony-project.org/wiki/SymfonyOnWampEnFrancais>.
- 7 Si tout va bien, vous devriez voir apparaître une liste de commandes associées à Symfony2.

Ces commandes vous serviront tout au long de la création de vos applications. C'est grâce à l'une de ces commandes que nous allons générer notre premier bundle.

Entrez la commande : **php app/console generate:bundle**. Le générateur vous demande alors de renseigner plusieurs options :

- **Bundle Namespace** : MyApp/FilmothequeBundle
- **Bundle Name** : MyAppFilmothequeBundle
- **Target Directory** : src
- **Configuration format** : yml
- **Create directory structure** : yes
- **Do you confirm generation** : yes
- **Confirm automatic update of your Kernel** : yes
- **Confirm automatic update of the Rooting** : yes

Une fois la génération terminée, vous devriez trouver un répertoire **MyApp** dans le dossier **/var/www/Symfony2/src/**. Ce répertoire contient plusieurs dossiers et plusieurs fichiers PHP que nous détaillerons plus tard.

Voilà, le bundle Filmotheque a été créé pour notre application **MyApp**. Vous noterez qu'il est nécessaire d'avoir le suffixe **Bundle** à la fin du nom de notre bundle **FilmothequeBundle**. Si vous avez besoin d'autres bundles pour votre application, alors il faudra les créer dans le répertoire **MyApp**.

IV-C - Lien entre le bundle et Symfony2

Sans vous en rendre compte, Symfony2 a été informé de la création de ce nouveau bundle au cours de sa génération. En répondant "yes" à la question "Confirm automatic update of your Kernel?", le générateur a mis à jour le noyau de Symfony2.

- Ouvrez le fichier **Symfony2/app/AppKernel.php**.
- Après la ligne :

```
new JMS\SecurityExtraBundle\JMSSecurityExtraBundle(),
```

Vous devriez voir le code :

```
new MyApp\FilmothequeBundle\MyAppFilmothequeBundle(),
```

Cette simple ligne vous permettra d'utiliser au sein de votre application toutes les fonctionnalités et tout le potentiel de Symfony2.

Pour plus de commodité, nous allons effectuer une petite modification dans l'un des fichiers de Symfony2. Elle nous permettra d'utiliser le préfixe **/myapp** dans les URL de notre application. Cela n'est pas nécessaire en dehors de ce tutoriel.

- Ouvrez le fichier **Symfony2/app/config/routing.yml**.
- Remplacez son contenu par le code suivant (attention à bien conserver l'indentation) :

```
MyAppFilmothequeBundle:
  resource: "@MyAppFilmothequeBundle/Resources/config/routing.yml"
  prefix:   /myapp
```

- Sauvegardez.

Symfony2 sait désormais que vous avez créé un nouveau bundle et notre application est prête à fonctionner. C'est ce que nous allons vérifier, en essayant d'afficher un premier message.

IV-D - Afficher un premier message

- Ouvrez le fichier **Symfony2/src/MyApp/FilmothequeBundle/Controller/DefaultController.php**.
- Dans la fonction appelée **indexAction()**, tapez le code :

```
public function indexAction()
{
    $message = 'Mon premier message';

    return $this->container-
>get('templating')->renderResponse('MyAppFilmothequeBundle:Default:index.html.twig',
    array('message' => $message)
    );
}
```

- Puis ouvrez le fichier **Symfony2/src/MyApp/FilmothequeBundle/Resources/views/index.html.twig**.
- Et insérez le texte suivant :

```
<p>{{ message }}</p>
```

- Ouvrez le fichier **Symfony2/src/MyApp/FilmothequeBundle/Resources/config/routing.yml**.
- Dans ce fichier, tapez le code :

```
myapp_accueil:
  pattern: /
  defaults: { _controller: MyAppFilmothequeBundle:Default:index }
```

- Dans votre navigateur web, tapez l'URL : http://localhost/Symfony2/web/app_dev.php/myapp/.
- Vous devriez voir le texte "Mon premier message" s'afficher.

V - Comprendre la structure des bundles

Quelques explications sont nécessaires pour comprendre ce que nous venons de faire.

Les contrôleurs (dossier "Controller")

Le fichier **DefaultController.php** est ce que l'on appelle un contrôleur. Ce fichier récupère les informations HTTP comme le détail de l'URL, les données de formulaire ou de session et affiche un message ou une page web. Ici, ce fichier contient une seule action qui est d'afficher le message "Mon premier message". Notre application a pour le moment un seul contrôleur et une seule action. À terme, elle aura plusieurs contrôleurs qui contiendront chacun plusieurs actions. Par exemple, nous aurons le contrôleur "ActeurController" qui gérera l'ajout, la modification et la liste des acteurs qui jouent dans les films. Par contre, nous n'utiliserons plus la fonction "echo" pour l'affichage mais des fichiers templates, c'est ce que nous verrons un peu plus loin.

Les routes (fichier "Resources/config/routing.yml")

Le fichier "routing.yml" fait le lien entre l'URL saisie dans le navigateur et l'action d'afficher le message. C'est ce que l'on appelle une route. Chaque route est un chemin qui mène à une destination. Pour un site web, le chemin correspond à une URL, et la destination est la page web à afficher ou l'action à réaliser. Ce fichier permet de centraliser l'ensemble des URL de notre application web et de les associer aux actions et aux pages à afficher. Ici, l'URL **myapp/** amènera systématiquement à l'action **indexAction()**. Nous définirons plus tard d'autres routes pour notre application. Par exemple, nous aurons une route **/ajouter-acteur** qui permettra de lancer l'action d'ajouter un acteur qui inclut la création du formulaire de saisie et la récupération des données à enregistrer.

Nous avons vu comment relier une URL à une action, ou plutôt une route à un contrôleur. Mais notre bundle contient d'autres dossiers, et il est nécessaire d'en créer de nouveaux :

Les vues (Dossier "Resources/view/")

Pour afficher les pages web, Symfony2 utilise un moteur de template baptisé Twig. Cela permet de séparer complètement le code PHP du code HTML, ce qui rend l'application plus claire et plus facile à maintenir. Le designer peut ainsi travailler en parallèle avec le développeur sur des fichiers bien distincts. Au sein de notre bundle, le dossier **Symfony2/MyApp/FilmothequeBundle/Resources/views/** a été créé. Tous les fichiers twig devront être classés dans ce répertoire, ce que nous ferons un peu plus tard.

Les entités (Dossier "Entity")

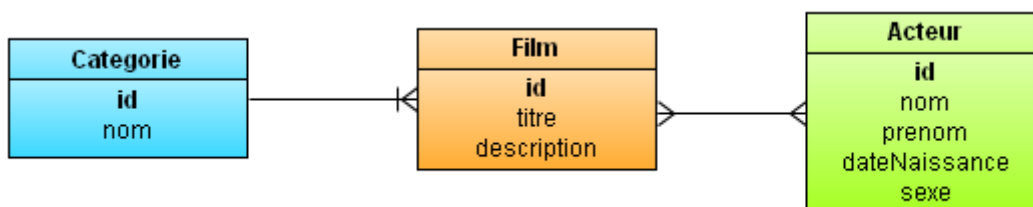
Au sein de votre bundle, créez un répertoire **Symfony2/MyApp/FilmothequeBundle/Entity**. Ce dossier n'est pas généré par défaut. Ce répertoire contiendra l'ensemble des entités. Une entité est une représentation objet d'un élément du monde réel. Nous aurons par exemple une entité "Categorie", une entité "Acteur" et une entité "Film" pour lesquelles nous définirons ce qu'elles contiennent. Un acteur aura un identifiant, un nom, un prénom, etc. La définition de chaque entité est très importante et il sera nécessaire de bien réfléchir à leur structure.

Les formulaires (Dossier "Form")

Au sein de votre bundle, créez un répertoire **Symfony2/MyApp/FilmothequeBundle/Form**. Ce dossier n'est pas généré par défaut. Ce répertoire contiendra l'ensemble des formulaires nécessaires à notre application. Nous aurons par exemple le formulaire d'ajout/modification des acteurs. Il ne s'agit pas de gérer l'ajout et la modification, ce qui sera déjà fait par le contrôleur, mais simplement de construire les éléments de notre formulaire (champs textes et libellés).

VI - Création des entités

Notre application doit gérer des films, des catégories et des acteurs. Les acteurs jouent dans les films et les films sont classés dans des catégories. Ces entités sont maintenant clairement identifiées et peuvent être représentées par le schéma suivant :



Soit sous forme de texte :

- Film : **id**, titre, description, categorie, acteurs ;
- Acteur : **id**, nom, prenom, date de naissance, sexe ;
- Categorie : **id**, nom.

Dans Symfony2, une entité est représentée par une classe d'objets. Nous allons ainsi créer plusieurs fichiers dans le répertoire "Entity" en prenant le soin de bien définir chaque élément.

Fichier Categorie.php

```
<?php
namespace MyApp\FilmothequeBundle\Entity;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Entity
 */
class Categorie
{
    /**
     * @ORM\GeneratedValue
     * @ORM\Id
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string",length="255")
     * @Assert\NotBlank()
     * @Assert\MinLength(3)
     */
    private $nom;
}
```

Fichier Acteur.php

```
<?php
namespace MyApp\FilmothequeBundle\Entity;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Entity
 */
class Acteur
{
    /**
     * @ORM\GeneratedValue
     * @ORM\Id
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string",length="255")
     * @Assert\NotBlank()
     * @Assert\MinLength(3)
     */
    private $nom;

    /**
     * @ORM\Column(type="string",length="255")
     * @Assert\NotBlank()
     * @Assert\MinLength(3)
     */
    private $prenom;

    /**
     * @ORM\Column(type="date")
     * @Assert\NotBlank()
     */
    private $dateNaissance;

    /**

```


Fichier Acteur.php

```
* @ORM\Column(type="string",length="1")
* @Assert\NotBlank()
* @Assert\Choice(choices = {"M", "F"})
*/
private $sexe;
}
```

Fichier Film.php

```
<?php
namespace MyApp\FilmothequeBundle\Entity;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Validator\Constraints as Assert;

/**
 * @ORM\Entity
 */
class Film
{
    /**
     * @ORM\GeneratedValue
     * @ORM\Id
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string",length="255")
     * @Assert\NotBlank()
     * @Assert\MinLength(3)
     */
    private $titre;

    /**
     * @ORM\Column(type="string",length="500")
     */
    private $description;

    /**
     * @ORM\ManyToOne(targetEntity="Categorie")
     * @Assert\NotBlank()
     */
    private $categorie;

    /**
     * @ORM\ManyToMany(targetEntity="Acteur")
     */
    private $acteurs;
}
```

Une fois ces fichiers créés, nous allons utiliser la console pour générer automatiquement les accesseurs, c'est-à-dire les fonctions **set()** et **get()** très souvent utilisées en programmation orientée objet. Nous pourrions le faire à la main, mais Symfony2 propose de le faire automatiquement, en quelques secondes...

Ouvrez tout d'abord le fichier **Symfony2/app/config/config.yml** et vérifiez que le paramètre **auto_mapping** a bien pour valeur **true** :

```
orm:
    auto_generate_proxy_classes: %kernel.debug%
    auto_mapping: true
```

Puis ouvrez la console et tapez la commande suivante : **php app/console doctrine:generate:entities MyApp**.

De nouveaux fichiers d'entités sont alors générés. Ils contiennent l'ensemble des fonctions **set()** et **get()** nécessaires au bon fonctionnement de l'application. Les anciens fichiers sont pour leur part renommés avec le suffixe

"~" (Acteur.php~, Categorie.php~ et Film.php~), ce qui vous permettra de récupérer les méthodes que vous auriez pu déjà écrire. Vous pouvez ensuite supprimer ces trois fichiers temporaires (ceux avec le suffixe "~"), ils ne serviront plus.

En fait, ces entités vont permettre de créer automatiquement les tables de la base de données. Pas besoin d'écrire de code SQL ni d'utiliser un logiciel spécifique ! Tout cela sera fait automatiquement grâce à Doctrine et les commentaires **@ORM...** qui définissent le type de champ. Les relations entre les entités, comme par exemple le fait qu'un film appartient à une catégorie, sont gérées par le commentaire "ManyToOne". Les relations multiples, comme le fait qu'un acteur peut jouer dans plusieurs films, sont décrites par le commentaire "ManyToMany".

Les commentaires de type **@Assert...** seront quant à eux utilisés lors de la soumission des formulaires, pour indiquer que tel ou tel champ ne doit pas être vide ou doit contenir au minimum trois caractères.

Il est donc important de bien définir vos entités et les commentaires associés. Je vous invite à lire la documentation de Symfony et de Doctrine pour avoir plus de détails et obtenir la liste exhaustive des valeurs **@ORM** et **@Assert** possibles.

VII - Création de la base de données et des tables

Jusqu'à présent, nous avons décrit notre application et nos entités, sans parler réellement de base de données. En effet, la présence de Doctrine dans Symfony2 permet d'omettre cette notion, et de nous intéresser à des objets plutôt qu'à des tables de base de données. Ainsi, nous pourrions très bien stocker nos films et acteurs dans des fichiers XML, dans une base SQLite ou MySQL, sans que cela ne change quoi que ce soit au code PHP que nous allons écrire. Doctrine s'occupe de faire l'interface avec la base de données et de l'interroger, ce qui nous permet d'utiliser uniquement des variables et des fonctions plutôt que d'écrire des requêtes SQL.

VII-A - Configurons notre base de données

- Ouvrez le fichier **Symfony2/app/config/parameters.ini**.
- Donnez le nom "filmotheque" à la base et choisissez le type **pdo_mysql** pour utiliser une base MySQL.
- Modifiez le code secret qui sera utilisé pour protéger votre application des attaques XSS.

```
[parameters]
database_driver  = pdo_mysql
database_host    = localhost
database_port    =
database_name    = filmotheque
database_user    = root
database_password =

mailer_transport = smtp
mailer_host      = localhost
mailer_user      =
mailer_password   =

locale          = fr

secret           = 16b10f9d2e7885152d41ea6175886563a
```

À noter que vous pouvez également utiliser l'assistant de configuration intégré dans Symfony2 en vous rendant à l'adresse : **http://localhost/Symfony2/web/app_dev.php/_configurator/**.

VII-B - Création de la base

- Ouvrez votre console.
- Tapez la commande : **php app/console doctrine:database:create**.
- Le message "Created database for connection named filmotheque" devrait s'afficher.

```
$ php app/console doctrine:database:create
Created database for connection named filmotheque
```

VII-C - Création des tables

- Ouvrez la console.
- Tapez la commande : **php app/console doctrine:schema:create**.
- Le message "Database schema successfully created" devrait s'afficher.

```
$ php app/console doctrine:schema:create
ATTENTION: This operation should not be executed in an production enviroment.
Creating database schema...
Database schema created successfully!
```

Vous pouvez vérifier que les tables ont été créées correctement (avec phpMyAdmin par exemple). Les noms de champ sont ceux que nous avons définis pour nos entités.

Si vous souhaitez ajouter une nouvelle entité ou avez effectué des modifications pour une entité (par exemple pour ajouter la date de sortie du film), alors il faudra mettre à jour les tables grâce à la commande : **php app/console doctrine:schema:update**.

Notre base de données est créée, nos tables aussi, voyons comment enregistrer une première donnée.

VII-D - Enregistrement d'une première donnée

Plutôt que d'effectuer une requête **INSERT INTO...**, nous allons utiliser la puissance de Symfony2 et de Doctrine pour ajouter des catégories de film.

- Ouvrez le fichier **MyApp\FilmothequeBundle\Controller\DefaultController.php**.
- En haut du fichier, remplacez les lignes par :

```
namespace MyApp\FilmothequeBundle\Controller;

use Symfony\Component\DependencyInjection\ContainerAware,
    Symfony\Component\HttpFoundation\RedirectResponse;
use MyApp\FilmothequeBundle\Entity\Categorie;

class DefaultController extends ContainerAware
```

- Puis remplacez la fonction **indexAction()** par le code suivant :

```
public function indexAction() {
    $em = $this->container->get('doctrine')->getEntityManager();

    $categorie1 = new Categorie();
    $categorie1->setNom('Comédie');
    $em->persist($categorie1);

    $categorie2 = new Categorie();
    $categorie2->setNom('Science-fiction');
    $em->persist($categorie2);

    $em->flush();

    $message = 'Catégories créées avec succès';
```

```
return $this->container->get('templating')->renderResponse('MyAppFilmothequeBundle:Default:index.html.twig',
    array('message' => $message)
);
```

- Ouvrez votre navigateur et rendez-vous sur la page http://localhost/Symfony2/web/app_dev.php/myapp/.
- Vérifiez dans votre base de données que les deux catégories ont bien été créées.
- Pour éviter de créer ces catégories chaque fois que vous accédez à cette URL, vous pouvez mettre la ligne **\$em->flush();** en commentaire ou bien renommer la fonction **indexAction()** pour **enregistrerCategorie()**.

VIII - Les templates Twig

Jusqu'à présent nous avons vu un peu de code PHP, un peu de fichiers de configuration, quelques commandes de la console mais nous n'avons pas encore réfléchi à l'affichage et la mise en page des données. Symfony2 intègre le moteur de template Twig que nous avons présenté un peu plus tôt. Vous avez peut-être l'habitude d'afficher vos variables PHP au milieu de votre code HTML comme ceci :

```
<p class="date"><?php echo $madate; ?></p>
```

Twig permet de simplifier cette écriture :

```
<p class="date">{{ madate }}</p>
```

Et si nous souhaitons afficher la date dans un format spécifique, pas besoin de code PHP :

```
<p class="date">{{ madate|date("d/m/Y") }}</p>
```

VIII-A - Création d'un premier template

Nous allons lister les catégories que nous avons créées précédemment.

- Ouvrez le fichier **MyApp\Filmotheque\Controller\DefaultController.php**.
- Remplacez la fonction **indexAction()** par :

```
public function indexAction()
{
    $em = $this->container->get('doctrine')->getEntityManager();
    $categories = $em->getRepository('MyAppFilmothequeBundle:Categorie')->findAll();

    return $this->container->get('templating')->renderResponse('MyAppFilmothequeBundle:Default:index.html.twig', array(
        'categories' => $categories
    ));
}
```

- Ouvrez ensuite le fichier **MyApp\Filmotheque\Resources\views\Default\index.html.twig**
- Tapez le code suivant

```
<html>
<body>
<h1>Liste des catégories</h1>
{% for cat in categories %}
    <p>{{ cat.nom }}</p>
{% else %}
```

```
<p>Aucune catégorie n'a été trouvée.</p>
{% endfor %}
</body>
</html>
```

- Allez à la page : http://localhost/Symfony2/web/app_dev.php/myapp/.
- Vous devriez voir les deux catégories "Comédie" et "Science-fiction" affichées.

Liste des catégories

Comédie

Science-fiction

Comme vous pouvez le voir, la liaison entre le code PHP et le template twig s'effectue grâce à la fonction **\$this->container->get('templating')->renderResponse()** dans laquelle nous définissons le fichier template utilisé et les variables à afficher.

Remarque : Si vous rencontrez des problèmes d'accentuation lors de l'affichage des catégories (exemple : lettre "é" remplacée par "?"), vérifiez que vos fichiers sont bien enregistrés au format UTF-8.

VIII-B - En savoir plus sur Twig

Twig est un outil très puissant et je vous invite à consulter sa documentation officielle pour découvrir toutes ses fonctionnalités : <http://www.twig-project.org>.

Si vous ne souhaitez pas utiliser Twig, sachez qu'il est possible avec Symfony2 d'avoir de simples fichiers HTML et d'afficher les variables classiquement avec la fonction PHP **echo**.

IX - Symfony2 et les contrôleurs

Pour gérer les acteurs, nous allons utiliser un nouveau contrôleur. Il nous permettra d'afficher une liste d'acteurs, d'ajouter et de modifier des acteurs, ou bien d'en supprimer. Commençons donc par créer un nouveau fichier **MyApp/FilmothequeBundle/Controller/ActeurController.php**. Dans ce fichier, créons les différentes actions dont nous avons besoin :

```
<?php

namespace MyApp\FilmothequeBundle\Controller;

use Symfony\Component\DependencyInjection\ContainerAware;
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;
use MyApp\FilmothequeBundle\Entity\Acteur;
//use MyApp\FilmothequeBundle\Form\ActeurForm;

class ActeurController extends ContainerAware
{
    public function listerAction()
    {
        return $this->container->get('templating')->renderResponse(
            'MyAppFilmothequeBundle:Acteur:lister.html.twig' );
    }

    public function ajouterAction()
    {
    }
```

```

return $this->container->get('templating')->renderResponse(
'MyAppFilmothequeBundle:Acteur:ajouter.html.twig');
}

public function modifierAction($id)
{
return $this->container->get('templating')->renderResponse(
'MyAppFilmothequeBundle:Acteur:modifier.html.twig');
}

public function supprimerAction($id)
{
return $this->container->get('templating')->renderResponse(
'MyAppFilmothequeBundle:Acteur:supprimer.html.twig');
}
}

```

Dans Symfony2, les actions peuvent avoir des paramètres. Ces paramètres proviennent des URL et des routes que nous allons définir juste après. De plus, toutes les actions d'un contrôleur se doivent de renvoyer vers une page. C'est pourquoi il est nécessaire d'utiliser **renderResponse()**. Il est également possible de faire une redirection comme nous le verrons plus tard.

N'oubliez pas de créer les différents fichiers templates twig définis dans le contrôleur **ActeurController.php**. Placez-les dans le dossier **/Resources/views/Acteur/**.

Relions maintenant chacune de ses actions à une URL. Pour cela ouvrez le fichier **MyApp/FilmothequeBundle/Resources/config/routing.yml** et ajoutez à la suite :

```

myapp_acteur_list:
  pattern: /acteur/
  defaults: { _controller: MyAppFilmothequeBundle:Acteur:liste }
myapp_acteur_ajouter:
  pattern: /acteur/ajouter
  defaults: { _controller: MyAppFilmothequeBundle:Acteur:ajouter }
myapp_acteur_modifier:
  pattern: /acteur/modifier/{id}
  defaults: { _controller: MyAppFilmothequeBundle:Acteur:modifier }
myapp_acteur_supprimer:
  pattern: /acteur/supprimer/{id}
  defaults: { _controller: MyAppFilmothequeBundle:Acteur:supprimer }

```

Vous noterez au passage :

- qu'il est possible de passer des informations dynamiques dans l'URL : nous passerons l'identifiant d'un acteur pour le modifier ou le supprimer grâce à l'utilisation de l'expression **{id}**. Nous aurions pu utiliser le prénom avec **{prenom}**. L'id est alors récupéré dans les paramètres de l'action du contrôleur ;
- que nous pourrions tout à fait utiliser des patterns d'URL différentes et par exemple, au lieu d'avoir **/acteur/ajouter**, nous aurions pu mettre **/ajouter-acteur**. Cette centralisation des URL permet de simplifier grandement leur maintenance ;
- que la valeur de **_controller** correspond aux actions que nous avons définies précédemment dans **ActeurController.php**, mais sans le suffixe **Action**.

Pour tester que vous n'avez pas fait d'erreur jusqu'ici, rendez-vous sur la page : **http://localhost/Symfony2/web/app_dev.php/myapp/acteur/**. Si une page blanche s'affiche, c'est que tout s'est bien déroulé. Vous pouvez ajouter du texte dans le template **views/Acteur/liste.html.twig** pour compléter votre test.

X - Symfony2 et les formulaires

X-A - Création d'un premier formulaire

Nous allons maintenant nous atteler à gérer l'ajout des acteurs et nous avons donc besoin pour cela de créer un formulaire.

- Créez le fichier **Form/ActeurForm.php**.
- Insérez le code suivant :

```
<?php

namespace MyApp\FilmothequeBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilder;

class ActeurForm extends AbstractType
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder
            ->add('nom')
            ->add('prenom')
            ->add('dateNaissance', 'birthday')
            ->add('sexe', 'choice', array('choices' => array('F'=>'Féminin', 'M'=>'Masculin')));
    }

    public function getName()
    {
        return 'acteur';
    }
}
```

Comme vous pouvez le voir, nous avons ajouté les différents champs dont nous avons besoin pour notre formulaire grâce à la fonction **add()**. Nous avons également nommé ce formulaire avec la fonction **getName()**. À aucun moment nous n'avons eu besoin d'écrire du code HTML pour insérer des champs de formulaire "input" ou "select", tout cela est géré automatiquement.

Il ne reste plus qu'à associer ce formulaire à notre action d'ajout et à l'afficher dans le template twig associé.

- Dans le contrôleur **ActeurController.php**, décommentez la ligne :

```
use MyApp\FilmothequeBundle\Form\ActeurForm;
```

- Puis modifiez la fonction **ajouterAction()** pour le code suivant :

```
public function ajouterAction()
{
    $acteur = new Acteur();
    $form = $this->container->get('form.factory')->create(new ActeurForm(), $acteur);

    return $this->container->get('templating')->renderResponse (
        'MyAppFilmothequeBundle:Acteur:ajouter.html.twig',
        array(
            'form' => $form->createView(),
            'message' => ''
        ));
}
```

- Ouvrez le fichier template **Acteur/ajouter.html.twig**.
- Insérez le code suivant :

```
<h1>Ajouter un acteur</h1>

{% if message %}
<p>{{ message }}</p>
{% endif %}

<form action="" method="post" {{ form_enctype(form) }}>
    {{ form_widget(form) }}
    <input type="submit" />
</form>
<p><a href="{{ path('myapp_acteur_list') }}">Retour à la liste des acteurs</a></p>
```

Rendez-vous sur la page http://localhost/Symfony2/web/app_dev.php/myapp/acteur/ajouter.

Le formulaire d'ajout d'acteur devrait s'afficher.

Ajouter un acteur

Nom

Prenom

Date naissance

Sexe

[Retour à la liste des acteurs](#)

Pour l'heure, le formulaire ne fait que s'afficher. Si vous essayez de saisir des valeurs et de soumettre le formulaire, aucune donnée n'est enregistrée en base de données. Cela va se faire en quelques lignes en modifiant l'action **ajouterAction()** pour :

```
public function ajouterAction()
{
    $message='';
    $acteur = new Acteur();
    $form = $this->container->get('form.factory')->create(new ActeurForm(), $acteur);

    $request = $this->container->get('request');

    if ($request->getMethod() == 'POST')
    {
        $form->bindRequest($request);

        if ($form->isValid())
        {
            $em = $this->container->get('doctrine')->getEntityManager();
            $em->persist($acteur);
            $em->flush();
            $message='Acteur ajouté avec succès !';
        }
    }

    return $this->container->get('templating')->renderResponse (
```



```
'MyAppFilmothequeBundle:Acteur:ajouter.html.twig',
array(
    'form' => $form->createView(),
    'message' => $message,
));
}
```

Vous remarquerez que nous n'avons vérifié aucun des champs soumis. Pourtant, si vous omettez le prénom de l'acteur, ou qu'il fait moins de trois caractères, un message d'erreur apparaîtra comme par magie... Si vous vous rappelez bien, lorsque nous avons créé les entités, nous avons inclus des commentaires de type **@Assert**. La validation des champs de formulaire s'effectue dès le départ lors de la définition des entités !

X-B - Le formulaire de modification

La modification des acteurs est très proche de l'ajout. La seule différence est qu'il faut récupérer les informations déjà stockées en base de données. Plutôt que d'avoir deux actions différentes, nous allons tout rassembler en une seule et même action "editer" :

Dans le fichier **MyApp/FilmothequeBundle/Resources/config/routing.yml**, remplacez les routes "ajouter" et "modifier" par :

```
myapp_acteur_ajouter:
    pattern: /acteur/ajouter
    defaults: { _controller: MyAppFilmothequeBundle:Acteur:editer }
myapp_acteur_modifieur:
    pattern: /acteur/modifier/{id}
    defaults: { _controller: MyAppFilmothequeBundle:Acteur:editer }
```

Les deux routes vont désormais utiliser la même action "editer" que nous allons maintenant créer dans le contrôleur :

```
public function editerAction($id = null)
{
    $message='';
    $em = $this->container->get('doctrine')->getEntityManager();

    if (isset($id))
    {
        // modification d'un acteur existant : on recherche ses données
        $acteur = $em->find('MyAppFilmothequeBundle:Acteur', $id);

        if (!$acteur)
        {
            $message='Aucun acteur trouvé';
        }
    }
    else
    {
        // ajout d'un nouvel acteur
        $acteur = new Acteur();
    }

    $form = $this->container->get('form.factory')->create(new ActeurForm(), $acteur);

    $request = $this->container->get('request');

    if ($request->getMethod() == 'POST')
    {
        $form->bindRequest($request);

        if ($form->isValid())
        {
            $em->persist($acteur);
            $em->flush();
        }
    }
}
```

```

    if (isset($id))
    {
        $message='Acteur modifié avec succès !';
    }
    else
    {
        $message='Acteur ajouté avec succès !';
    }
}

return $this->container->get('templating')->renderResponse(
'MyAppFilmothequeBundle:Acteur:editer.html.twig',
array(
'form' => $form->createView(),
'message' => $message,
));
}

```

Pensez à renommer le fichier template **ajouter.html.twig** pour **editer.html.twig**, sinon vous allez obtenir un beau message d'erreur !

Si vous obtenez tout de même un message d'erreur, il vous faut peut-être rafraîchir le cache de Symfony en tapant la commande **php app/console cache:clear**.

XI - Création d'une page web simple

Passons maintenant à la création de la liste des acteurs. Le principe reste le même : créer l'action "listAction" et le template twig associé.

Insérez le code suivant dans le fichier **ActeurController.php** :

```

public function listAction()
{
    $em = $this->container->get('doctrine')->getEntityManager();

    $acteurs= $em->getRepository('MyAppFilmothequeBundle:Acteur')->findAll();

    return $this->container-
>get('templating')->renderResponse('MyAppFilmothequeBundle:Acteur:li
array(
'acteurs' => $acteurs
));
}

```

Fichier lister.html.twig

```

<h1>Liste des acteurs</h1>

<table>
{% for a in acteurs %}
<tr>
<td>{{ a.nom }}</td>
<td>{{ a.prenom }}</td>
<td>{{ a.dateNaissance|date('d/m/Y') }}</td>
<td>{{ a.sexe }}</td>
<td><a href="{{ path('myapp_acteur_modifier', { 'id': a.id }) }}">Modifier</a></td>
<td><a href="{{ path('myapp_acteur_supprimer', { 'id': a.id }) }}">Supprimer</a></td>
</tr>
{% else %}
<tr><td>Aucun acteur n'a été trouvé.</td></tr>
{% endfor %}
</table>

<p><a href="{{ path('myapp_acteur_ajouter') }}">Ajouter un acteur</a><p>

```

Si vous avez ajouté quelques acteurs, voici le résultat que vous devriez obtenir :

Liste des acteurs

Reno Jean 30/07/1948 M [Modifier](#) [Supprimer](#)

Deneuve Catherine 22/10/1943 F [Modifier](#) [Supprimer](#)

[Ajouter un acteur](#)

XII - Supprimer des données

La gestion des acteurs est presque terminée. Nous avons un formulaire d'ajout et de modification, ainsi que la liste des acteurs enregistrés. Il ne manque plus que la suppression qui va s'écrire en quelques lignes dans notre contrôleur :

```
public function supprimerAction($id)
{
    $em = $this->container->get('doctrine')->getEntityManager();
    $acteur = $em->find('MyAppFilmothequeBundle:Acteur', $id);

    if (!$acteur)
    {
        throw new NotFoundException("Acteur non trouvé");
    }

    $em->remove($acteur);
    $em->flush();

    return new RedirectResponse($this->container->get('router')->generate('myapp_acteur_lister'));
}
```

Après avoir récupéré l'acteur et vérifié qu'il existait bien, nous le supprimons grâce à **remove()**. Puis, on redirige l'utilisateur vers la liste des acteurs.

Voilà, la gestion des acteurs est terminée. Depuis le début nous avons écrit assez peu de code PHP. Imaginez tout le code que vous auriez dû écrire pour faire la même chose sans Symfony2. C'est la grande force de ce framework : écrire le moins possible, ne pas se répéter, et avoir une organisation propre et logique.

XIII - À vous de jouer !

Saurez-vous finir notre application en réalisant la gestion des catégories et celles des films ? La démarche est identique à celle des acteurs... Un contrôleur, un formulaire, des fichiers Twig... À vous de jouer !

Pour les plus pressés, une des solutions possibles se trouve dans ce fichier : [MyApp.zip](#).

Vous pourrez poursuivre votre apprentissage de Symfony2 avec la suite de ce tutoriel intitulée : [Améliorez vos applications développées avec Symfony2](#).

XIV - Références

- Site officiel de Symfony : www.symfony.com
- Site officiel de Twig : www.twig-project.org
- Site officiel de Doctrine : www.doctrine-project.org

XV - Remerciement

Je tiens à remercier **dourouc05**, **nathieb** et **creativecwx** pour leurs conseils lors de la rédaction de ce tutoriel ainsi que **ClaudeLELOUP** pour sa relecture.