

Améliorez vos applications développées avec Symfony2

par Jérôme Place [jpcheck](#)

Date de publication : 4 juin 2011

Dernière mise à jour : 10 août 2011


Ce tutoriel sur le framework Symfony2 aborde les notions de sécurité, d'Ajax, de traduction, de mise en page et de personnalisation des formulaires à travers le développement d'une application web.

I - Introduction.....	3
II - Améliorer la mise en page.....	3
II-A - Les Templates twig et l'héritage.....	3
II-B - Utiliser un template dans un autre template.....	6
II-C - Intégrer le résultat d'un contrôleur dans un template.....	6
II-D - Insérer des images et appliquer des styles CSS.....	7
III - Améliorer les formulaires.....	9
III-A - Des champs et des labels personnalisés.....	9
III-B - Personnaliser une liste de choix.....	10
III-C - Changer la disposition des champs.....	10
III-D - Des messages d'erreur personnalisés.....	11
IV - Symfony2 et la méthode Ajax.....	12
V - Une interface multilingue.....	15
V-A - Choix de la langue avec setLocal().....	15
V-B - Activation de la traduction.....	16
V-C - Création de fichiers de langue.....	17
V-D - Traduction dans les templates twig avec la fonction trans().....	17
V-E - Traduction des labels des champs de formulaire.....	18
V-F - Traduction des messages d'erreur.....	18
V-G - Traduction au niveau des contrôleurs.....	18
VI - Utilisateurs et espace sécurisé.....	19
VI-A - La gestion des utilisateurs avec FOSUserBundle.....	20
VI-A-1 - Installation de FOSUserBundle.....	20
VI-A-2 - Utiliser FOSUserBundle au sein de l'application.....	21
VI-A-3 - Créer un premier utilisateur.....	22
VI-A-4 - Intégration des routes et des templates de FOSUserBundle.....	22
VI-B - Gestion des droits d'accès.....	23
VI-B-1 - Activer la sécurité.....	23
VI-B-2 - Accès au formulaire de connexion.....	24
VI-B-3 - Configurer les pages aux accès restreints.....	25
VII - Conclusion.....	26
VIII - Références.....	27
IX - Remerciements.....	27

I - Introduction

Ce tutoriel fait suite à l'article intitulé "**Créer sa première application web en PHP avec Symfony2**" au cours duquel vous avez pu découvrir le framework Symfony2 et ses grands principes de fonctionnement. Si vous ne l'avez pas encore lu, je vous invite à le parcourir dès maintenant car nous allons poursuivre le développement de l'application "Filmothèque" créée précédemment.

Au cours de cet article, nous allons voir comment améliorer notre application pour parvenir à un site web complet et prêt à être mis en ligne. Nous travaillerons sur la mise en page grâce à l'héritage des templates Twig, à l'insertion d'images et de styles CSS. Nous perfectionnerons les formulaires avec des labels, des messages d'erreur et une disposition personnalisés. Nous aborderons ensuite les notions d'Ajax pour augmenter l'interactivité utilisateur et de traduction pour une interface multilingue. Enfin, nous verrons comment intégrer un bundle tiers pour gérer des utilisateurs et sécuriser les formulaires de notre application avec un formulaire de connexion.

Ce tutoriel est basé sur la version finale Symfony2.0.0 que vous pouvez télécharger ici :  <http://symfony.com/download>.

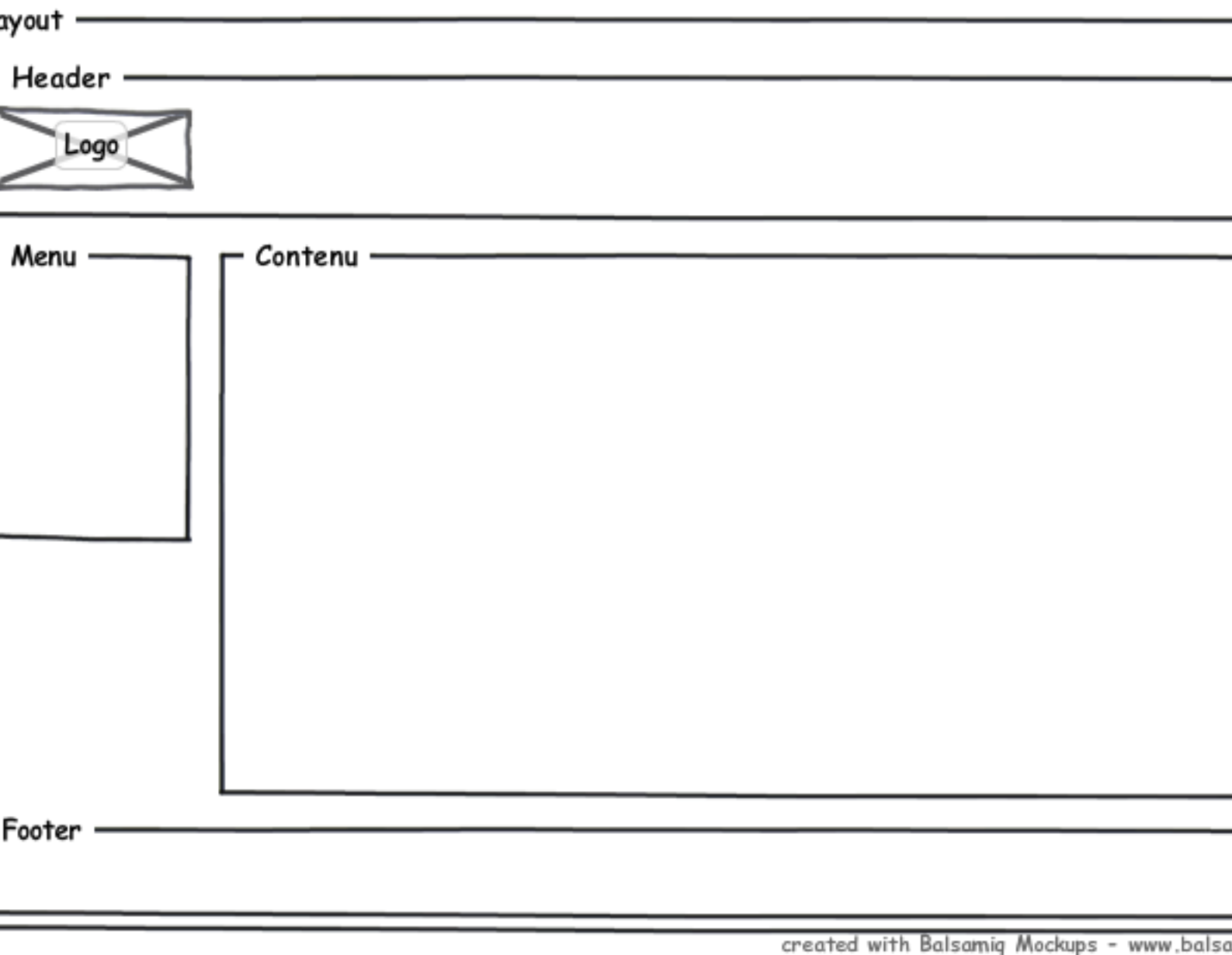
II - Améliorer la mise en page

Qu'il s'agisse d'un site internet, d'un intranet ou même d'un jeu en ligne, une application web doit avoir sa propre identité et garder une certaine cohérence entre ses pages : présence d'un logo, mise en place d'un ou plusieurs menus, utilisation de couleurs et de polices de caractères similaires, etc. L'utilisateur peut ainsi naviguer facilement entre les pages et retrouver rapidement les informations dont il a besoin. L'interface graphique et la mise en page sont donc des éléments à ne pas négliger si vous souhaitez avoir une application claire et intuitive. Nous allons voir comment améliorer la mise en page de notre application Filmothèque réalisée avec Symfony2.

II-A - Les Templates twig et l'héritage

Comme nous l'avons vu lors du tutoriel précédent, Twig permet d'insérer du texte et d'intégrer des variables issues de PHP grâce à l'utilisation de doubles accolades (ex : `{{ mon_message }}`). Twig va beaucoup plus loin et permet d'inclure tout ou une partie de templates twig existants. Ainsi, vous allez pouvoir retrouver sur l'ensemble de vos pages des éléments similaires comme un logo, des menus, un en-tête (header), un pied de page (footer), etc. Cela est possible grâce à la notion d'héritage qui existe dans Twig. Le template parent va contenir les éléments communs à chaque page (ex : logo, menu, etc.) et le template enfant va contenir les éléments spécifiques à la page recherchée (ex : actualités, fiche de films, formulaire, etc.).

Le schéma suivant propose une interface assez simple où les blocs "header", "menu" et "footer" seront communs à chaque page, alors que le bloc "contenu" variera en fonction de la page.



created with Balsamiq Mockups - www.balsamiq.com

En pratique, il va falloir créer un template parent qui contiendra tous les éléments en commun. Nous allons l'appeler "layout". Ce template aura également une zone modifiable, identifiée grâce au mot-clé "block" dont le contenu pourra varier en fonction du template enfant qui est appelé. Au niveau des templates enfants, il sera nécessaire de préciser le template qui doit être hérité en utilisant le mot-clé "extends" suivi du nom du template parent. Enfin, le contenu de la zone modifiable devra être défini dans chaque template enfant grâce au mot-clé "block" suivi du nom de la zone. Voyons tout cela autour d'un exemple :

Commencez par créer le fichier **MyApp/FilmothequeBundle/Resources/views/layout.html.twig** et insérez le code suivant :

```
<html>
<head>
  <title>{% block title %}Filmothèque{% endblock %}</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
</head>
<body>
  <div id="page">
```

```

        <div id="header">
<ul id="menu">
    <li><a href="{{ path('myapp_accueil') }}">Accueil</a></li>
    <li><a href="{{ path('myapp_film_lister') }}">Films</a></li>
    <li><a href="{{ path('myapp_acteur_lister') }}">Acteurs</a></li>
</ul>
        </div>

        <div id="content">
            <h1>{{ block('title') }}</h1>
            {% block content %}
            {% endblock %}
        </div>
        <div id="footer">
            <p>Tous droits réservés © 2011</p>
        </div>
    </div>
</body>
</html>

```

Nous avons défini ici deux zones de contenu dynamique :

- **title**, qui fait référence au titre de la page (balise title). Le contenu de ce bloc est répété dans la balise h1 grâce à la fonction block().
- **content**, qui correspond au contenu spécifique de la page

Les sections "menu", "header" et "footer" sont quant à elles des éléments statiques qui resteront les mêmes quelle que soit la page affichée.

Vous remarquerez qu'une zone dynamique peut avoir une valeur par défaut qui sera affichée si aucune valeur n'est définie dans le template enfant. C'est le cas du bloc "title" qui affichera par défaut le texte "Filmothèque". Par ailleurs, comme les noms des blocs doivent être uniques, il existe la fonction **block()** qui permet de recopier si nécessaire le contenu d'un bloc existant. Dans notre exemple, la balise h1 contiendra la même valeur que le bloc titre. Sachez enfin que vous pouvez créer autant de zones **{% block %}** que nécessaire et avoir plusieurs templates parents si besoin.

Voyons maintenant comment se présentent les templates enfants.

Modifiez le template **views/Acteur/lister.html.twig** et insérez le code suivant :

```

{% extends 'MyAppFilmothèqueBundle::layout.html.twig' %}

{% block title %}Liste des acteurs{% endblock %}

{% block content %}
<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>
{% endblock %}

```

Ce template hérite de **MyAppFilmothequeBundle::layout.html.twig** grâce au mot-clé **extends**. Le contenu des blocs "title" et "content" est également défini. Ainsi, si vous allez sur la page http://localhost/Symfony2/web/app_dev.php/myapp/acteur/, vous devriez voir apparaître tous les éléments communs (le menu, le footer) ainsi que le contenu propre de la page. Répétez cette opération pour l'ensemble des templates enfants afin qu'ils héritent tous du fichier layout.

II-B - Utiliser un template dans un autre template

L'héritage de Twig autorise une communication dans les deux sens. Ainsi, un template parent peut à son tour transmettre des informations à un template enfant. Cela est pratique si vous savez qu'un template va être utilisé à différents endroits de votre application. Par exemple, une liste d'éléments aura la même présentation que l'on soit sur les pages "Top éléments", "Éléments archivés" ou encore "Tous les éléments". Nous allons mettre cela en pratique avec la liste des acteurs qui aura un format identique que l'on soit sur la page "Top acteur", "Top actrice" ou "Tous les acteurs". Nous allons avoir un template de base, que nous appellerons "liste", qui pourra être inclus dans plusieurs templates parents.

Créez le fichier **views/Acteur/liste.html.twig** et insérez le code :

```
<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>
    {% else %}
        <tr>
            <td>Aucun acteur n'a été trouvé.</td>
        </tr>
    {% endfor %}
</table>
```

Puis, remplacez le contenu du fichier **Acteur/liste.html.twig** par :

```
{% extends 'MyAppFilmothequeBundle::layout.html.twig' %}

{% block title %}Liste des acteurs{% endblock %}

{% block content %}

{% include 'MyAppFilmothequeBundle:Acteur:liste.html.twig' with {'acteurs' : acteurs} %}

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

{% endblock %}
```

L'intégration du template enfant s'effectue par la fonction "include" à laquelle on peut passer plusieurs paramètres qui proviennent du contrôleur parent. Dans notre exemple, la variable "acteurs" est récupérée et transmise au template "liste".

Vous pouvez reproduire la même chose avec la liste des films.

II-C - Intégrer le résultat d'un contrôleur dans un template

Si vous souhaitez intégrer la liste des acteurs dans l'une de vos pages, il faut utiliser la fonction "render". Cette fonction très pratique n'inclut pas seulement un template comme nous l'avons ci-dessus mais fait appel directement

au contrôleur. Celui-ci récupère les informations de la base de données, effectue ses calculs, et renvoie le résultat sous forme de code HTML. Il faut néanmoins faire attention à ce que le contrôleur appelé utilise un template simple, qui n'hérite pas du layout principal, sinon vous risquez de voir apparaître plusieurs fois votre menu... Cela va nous permettre d'afficher la liste des acteurs les plus jeunes et la liste des films sur la page d'accueil de notre application.

Ouvrez le fichier **Controller/ActeurController.php** et insérez une nouvelle fonction **topAction()** :

```
public function topAction($max = 5)
{
    $em = $this->container->get('doctrine')->getEntityManager();

    $qbm = $em->createQueryBuilder();
    $qbm->select('a')
        ->from('MyAppFilmothequeBundle:Acteur', 'a')
        ->orderBy('a.dateNaissance', 'DESC')
        ->setMaxResults($max);

    $query = $qbm->getQuery();
    $acteurs = $query->getResult();

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

Cette nouvelle fonction permet d'afficher le top 5 des acteurs les plus jeunes (classés par leur date de naissance). Elle utilise le template **liste.html**, qui n'hérite d'aucun autre template. Vous pouvez réaliser la même chose en créant le "Top Film" de votre choix.

Puis, ouvrez le template de la page d'accueil (fichier **Default/index.html.twig**) et ajoutez :

```
{% block content %}
<h3>Top des acteurs les plus jeunes</h3>
{% render "MyAppFilmothequeBundle:Acteur:top" with {'max': 3} %}

<h3>Top films</h3>
{% render "MyAppFilmothequeBundle:Film:top" %}

{% endblock %}
```

Voilà, la page d'accueil affiche désormais deux listes qui sont générées par différents contrôleurs. En effet, nous n'avons pas touché au contrôleur de la page d'accueil (DefaultController.php) et pourtant nous avons pu intégrer du contenu dynamique. À noter que nous aurions très bien pu utiliser cette fonction "render" sur le template principal "layout", ce qui aurait permis d'afficher le top acteur sur toutes les pages de notre application (dans une colonne à droite par exemple).

II-D - Insérer des images et appliquer des styles CSS

Les images, tout comme les styles CSS, participent à l'amélioration de l'interface graphique et de la mise en page d'une application. Dans Symfony2, les images et les styles CSS sont des ressources propres à un bundle, ce qui signifie que chaque bundle peut avoir ses propres styles CSS et ses propres images. Ainsi, ces éléments sont créés non pas au niveau de l'application mais au niveau des bundles, comme c'est déjà le cas pour les templates (répertoire "views"). Images, feuilles CSS et autres fichiers JavaScript d'un bundle devront être copiés dans le répertoire "public". Ce dossier contiendra tous les éléments qui seront accessibles par tous. Il n'est pas généré par défaut et doit donc être créé : **MyApp/FilmothequeBundle/Resources/public/**

Pour séparer les différents éléments, vous pouvez créer des sous-dossiers : **"images/"** qui contiendra les images utilisées dans l'application, et **"css/"** qui contiendra l'ensemble des feuilles de styles CSS.

Voyons comment intégrer un logo et associer une feuille de style aux templates de notre application Filmothèque.

Copiez l'image **logo.png** (à télécharger ci-dessous) dans le dossier **MyApp/FilmothèqueBundle/Resources/public/images/**



Copiez maintenant la feuille de styles **main.css** (à télécharger [ici](#)) dans le dossier **MyApp/FilmothèqueBundle/Resources/public/css/**

Puis, ouvrez le template principal **views/layout.html.twig** et insérez les lignes suivantes :

```
// ...
<link rel="stylesheet" href="{ asset('bundles/myappfilmothèque/css/main.css') }}" type="text/css" media="all" />
</head>
<body>
    <div id="page">
<div id="header">

// ...
```

La fonction **asset()** permet d'appeler un fichier image, CSS ou bien JavaScript. Le dossier **public/** de notre bundle n'est pas accessible directement : si vous cherchez à recharger une page de l'application vous ne verrez pas le logo apparaître. Il est en effet nécessaire de "publier" le contenu public des bundles dans le répertoire **web/**. Seul ce répertoire est accessible par les visiteurs du site. C'est d'ailleurs ce répertoire qui contient le fichier **app_dev.php** auquel nous faisons appel.

Pour publier les éléments publics des différents bundles, ouvrez la console puis tapez la commande : **php app/console assets:install web**

Les fichiers images et CSS de notre bundle filmothèque sont alors copiés vers le dossier **web/bundles/myappfilmothèque/**.

Rechargez l'application dans votre navigateur. Vous devriez maintenant voir le logo et la nouvelle mise en page avec les styles CSS.



ACCUEIL FILMS ACTEURS

Accueil

Top des acteurs les plus jeunes

Portman	Natalie	09/05/1981	F	Modifier	Supprimer
Dujardin	Jean	19/06/1972	M	Modifier	Supprimer
Reno	Jean	31/07/1948	M	Modifier	Supprimer

Top films

Brice de Nice	Voir	Modifier	Supprimer
Léon	Voir	Modifier	Supprimer
Le Dernier Métro	Voir	Modifier	Supprimer

Vous pouvez bien entendu intégrer d'autres images et adapter les styles CSS selon vos goûts. Attention à modifier les fichiers du bundle et pas ceux du répertoire **web/**. Une fois changés, publiez-les à nouveau en relançant la commande **assets:install**.

III - Améliorer les formulaires

Après la mise en page de notre application, nous allons revenir sur nos formulaires pour les personnaliser et améliorer leur présentation.

III-A - Des champs et des labels personnalisés

Lorsque nous avons créé notre formulaire d'ajout d'acteur (voir tutoriel précédent), nous n'avons pas eu besoin de préciser le type de champ (ex : "input text") ni le label (ex : Nom de famille) dont nous avons besoin. Pourtant, Symfony2 a pu générer un formulaire par défaut à l'aide des informations fournies par chaque entité. Heureusement, il est possible de personnaliser à la fois le type de champ et le label de chaque élément du formulaire. Cela s'effectue lors de l'ajout de l'élément, grâce à l'option **label** de la fonction **add()**.

Fichier Form/ActeurForm.php

```
// ...
$builder->add('nom', 'text', array('label' => 'Nom de famille'))
->add('prenom', 'text', array('label' => 'Prénom'))
->add('dateNaissance', 'birthday', array('label' => 'Date de naissance'));
// ...
```

Ici, notre champ "nom" aura pour libellé "Nom de famille" et correspondra à un champ de texte classique. La date de naissance sera quant à elle de type "birthday", ce qui permettra de choisir une année de naissance entre 1890 et aujourd'hui (environ 120 ans).

Pour chaque élément de formulaire, vous pourrez ainsi choisir le type de champ à afficher. Vous pourrez les personnaliser grâce aux nombreuses options disponibles. Vous trouverez la liste des champs prédéfinis dans Symfony2 sur la page suivante : <http://symfony.com/doc/current/reference/forms/types.html>.

III-B - Personnaliser une liste de choix

Si, au sein d'un formulaire, vous avez une liste qui fait référence à une entité, comme par exemple la liste des acteurs qui participent à un film, vous pouvez tout à fait afficher le nom et le prénom de chaque acteur plutôt que d'afficher seulement le nom de famille.

Ouvrez le fichier **Entity/Acteur.php** et insérez une nouvelle fonction **getNomPrenom()** :

```
public function getPrenomNom()
{
    return $this->prenom.' '.$this->nom;
}
```

Modifiez ensuite l'option **property** du champ Acteur dans le formulaire des films :

Fichier Form/FilmForm.php

```
// ...
$builder->add('acteurs', 'entity', array(
    'class' => 'MyApp\FilmothequeBundle\Entity\Acteur',
    'property' => 'PrenomNom',
    'expanded' => true,
    'multiple' => true,
    'required' => false
));
// ...
```

Allez sur la page http://localhost/Symfony2/web/app_dev.php/myapp/film/ajouter : le prénom et le nom des acteurs sont désormais visibles. En fait, Symfony va vérifier si le champ PrenomNom ou la méthode **getPrenomNom()** existe bien et récupère son résultat pour l'affichage du champ.

III-C - Changer la disposition des champs

Par défaut, Symfony2 et Twig génèrent un formulaire dont les éléments (labels, champs) sont disposés les uns à la suite des autres. Cette disposition est suffisante dans la plupart des cas, mais vous pouvez tout à fait agencer votre formulaire comme vous le désirez. Pour afficher un formulaire dans un template nous avons utilisé la formule courte :

```
{{ form_widget(form) }}
```

Pour changer la disposition, il va falloir déclarer les éléments un par un. On peut par exemple imaginer une disposition en plusieurs colonnes pour notre formulaire d'acteur :

Fichier Acteur/editer.html.twig

```
{% if message %}
<p class="message">{{ message }}</p>
{% endif %}

<form action="" method="post" {{ form_enctype(form) }}>
<div class="error">
    {{ form_errors(form) }}
</div>
<table>
    <tr>
```

Fichier Acteur/editer.html.twig

```
<td>{{ form_label(form.nom) }}</td>
<td>{{ form_label(form.prenom) }}</td>
    <td>{{ form_label(form.dateNaissance) }}</td>
    <td>{{ form_label(form.sexe) }}</td>
</tr>
<tr>
<td>
    {{ form_errors(form.nom) }}
    {{ form_widget(form.nom) }}
</td>
<td>
    {{ form_errors(form.prenom) }}
    {{ form_widget(form.prenom) }}
</td>
<td>
    {{ form_errors(form.dateNaissance) }}
    {{ form_widget(form.dateNaissance) }}
</td>
<td>
    {{ form_errors(form.sexe) }}
    {{ form_widget(form.sexe) }}
</td>
</tr>
</table>

{{ form_rest(form) }}

<input type="submit" />
</form>
```

Nous avons utilisé :

- **form_label()** qui permet d'afficher le libellé d'un champ ;
- **form_error()** qui permet d'afficher les erreurs éventuelles d'un champ ;
- **form_widget()** qui permet d'afficher le champ lui-même.

Vous pouvez également utiliser **form_row()** qui permet d'afficher ces trois éléments d'un coup.

La fonction **form_rest()** va quant à elle générer les champs restants du formulaire, c'est-à-dire tous ceux que vous n'avez pas insérés manuellement (si vous réorganisez seulement une partie du formulaire). Cette fonction est nécessaire même si vous avez ajouté tous les champs, car elle va aussi intégrer les champs cachés (comme le token) qui sont nécessaires au bon fonctionnement du formulaire.

Vous pouvez maintenant accéder à la page http://localhost/Symfony2/web/app_dev.php/myapp/acteur/ajouter. Vous devriez obtenir le résultat suivant :

Ajouter/Modifier un acteur

Nom de famille

Prénom

Date de naissance

Sexe

Valider

III-D - Des messages d'erreur personnalisés

En définissant nos entités (dossier "Entity/"), nous avons introduit plusieurs règles de validation des formulaires. Par exemple, si l'utilisateur saisit moins de trois caractères pour le nom de famille d'un acteur, alors le formulaire est considéré comme invalide et un message d'erreur apparaît. Par défaut, les messages d'erreur sont en anglais et il

s'avère donc nécessaire de les personnaliser. Nous allons pouvoir le faire assez simplement au niveau de chaque règle de validation.

Ouvrez le fichier d'entité **Acteur.php** et remplacez l'annotation suivante :

```
@Assert\MinLength(3)
```

Par :

```
@Assert\MinLength(limit = 3, message = "Le nom de famille doit avoir au moins {{ limit }} caractères")
```

Vous pouvez maintenant tester le formulaire d'ajout d'acteur en ne saisissant que deux lettres pour le nom de famille et vérifiez que ce nouveau message d'erreur s'affiche bien. Si cela ne fonctionne pas, c'est peut-être que le cache interne de Symfony2 n'est pas à jour. Pour le nettoyer, ouvrez la console et tapez la commande : **php app/console cache:clear**

Ajouter/Modifier un acteur

Nom de famille

Le nom de famille doit avoir au moins 3 caractères

En fait, toutes les règles de validation possèdent un attribut "message" qui correspond au message d'erreur à afficher si la contrainte n'est pas respectée. Vous remarquez que le message peut intégrer un autre attribut comme c'est le cas pour {{ limit }} qui sera remplacé par sa valeur lors de l'affichage (ici le chiffre 3).

Vous allez ainsi pouvoir reprendre chaque annotation **@Assert** pour lui associer le message d'erreur de votre choix. À noter que vous trouverez toutes les contraintes prédéfinies dans Symfony2 sur la page suivante : <http://symfony.com/doc/current/reference/constraints.html>.

IV - Symfony2 et la méthode Ajax

Très utilisée dans les applications dites "web 2.0", Ajax est une méthode qui permet de modifier partiellement une page web pour la mettre à jour sans avoir à recharger la page entière. Seules les informations utiles sont renouvelées ce qui limite les échanges avec le serveur et rend l'interface plus rapide et plus réactive.

Nous allons voir comment mettre en place un moteur de recherche d'acteurs avec cette méthode. Nous aurons un formulaire comprenant un champ texte de saisie d'un mot-clé et un bouton rechercher. La méthode Ajax sera utilisée lors de la soumission du formulaire pour ne mettre à jour que les résultats de la recherche.

Créez le fichier **ActeurRechercheForm.php** et placez-le dans le répertoire **Form/**

Insérez le code :

```
<?php

namespace MyApp\FilmothequeBundle\Form;

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

class ActeurRechercheForm extends AbstractType
```

```
{
    public function buildForm(FormBuilder $builder, array $options)
    {
        $builder->add('motcle', 'text', array('label' => 'Mot-clé'));
    }

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

Puis, ouvrez le fichier **Controller/ActeurController.php** et ajoutez en entête la ligne :

```
use MyApp\FilmothequeBundle\Form\ActeurRechercheForm;
```

Modifiez la fonction **listAction()** pour ajouter le formulaire :

```
public function listAction()
{
    // ...
    $form = $this->container->get('form.factory')->create(new ActeurRechercheForm());

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

Changez enfin le template **Acteur/liste.html.twig** pour afficher le formulaire de recherche :

```
{% block content %}
<form id="form_recherche" action="{{ path('myapp_acteur_rechercher') }}" method="post">
    {{ form_widget(form) }}
    <input type="submit" value="Rechercher" />
</form>

//...
{% endblock %}
```

Notre formulaire est en place, il faut maintenant l'associer à une action qui s'occupera de rechercher les acteurs en fonction du mot-clé saisi.

Dans le fichier **ActeurController.php**, créez la fonction **rechercherAction()** :

```
public function rechercherAction()
{
    $request = $this->container->get('request');

    if($request->isXmlHttpRequest())
    {
        $motcle = '';
        $motcle = $request->request->get('motcle');

        $em = $this->container->get('doctrine')->getEntityManager();

        if($motcle != '')
        {
            $qb = $em->createQueryBuilder();

            $qb->select('a')
```

```

->from('MyAppFilmothequeBundle:Acteur', 'a')
->where("a.nom LIKE :motcle OR a.prenom LIKE :motcle")
->orderBy('a.nom', 'ASC')
->setParameter('motcle', '%'.$motcle.'%');

$query = $qb->getQuery();
$acteurs = $query->getResult();
}
else {
    $acteurs = $em->getRepository('MyAppFilmothequeBundle:Acteur')->findAll();
}

return $this->container-
>get('templating')->renderResponse('MyAppFilmothequeBundle:Acteur:liste.html.twig', array(
    'acteurs' => $acteurs
));
}
else {
    return $this->listAction();
}
}

```

Cette fonction va vérifier si la méthode Ajax a été utilisée (isXmlHttpRequest). Si tel est le cas, alors elle récupère le mot-clé en provenance du formulaire et recherche les acteurs dont le nom ou le prénom est similaire (LIKE '%...%'). Sinon, elle retourne la liste complète des acteurs.

Ouvrez le fichier **routing.yml** pour créer une nouvelle route vers cette action :

```

myapp_acteur_rechercher:
    pattern: /acteur/rechercher
    defaults: { _controller: MyAppFilmothequeBundle:Acteur:rechercher }
    requirements:
        _method: POST

```

Vous noterez que nous avons restreint l'accès à cette page puisque seule une soumission de formulaire par la méthode POST permettra d'y accéder (requirement method : POST).

Dans la partie **head** du fichier **views/layout.html.twig**, ajoutez la librairie JavaScript **JQuery** qui simplifiera l'emploi de la méthode Ajax :

```

<head>
// ...
<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.5.1/jquery.min.js" type="text/javascript"></script>
</head>

```

Puis, modifiez le code du fichier **Acteur/lister.html.twig** pour :

```

{% block content %}

<form id="form_recherche" action="{{ path('myapp_acteur_rechercher') }}" method="post">
    {{ form_widget(form) }}
    <input type="submit" value="Rechercher" />
</form>
<div class="loading"></div>
<div id="resultats_recherche">
    {% include 'MyAppFilmothequeBundle:Acteur:liste.html.twig' with {'acteurs' : acteurs} %}
</div>

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

<script>
$(".loading").hide();

```

```
$("#form_recherche").submit(function() {
    $(".loading").show();
    var motcle = $("#acteurrecherche_motcle").val();
    var DATA = 'motcle=' + motcle;
    $.ajax({
        type: "POST",
        url: "{{ path('myapp_acteur_rechercher') }}",
        data: DATA,
        cache: false,
        success: function(data) {
            $('#resultats_recherche').html(data);
            $(".loading").hide();
        }
    });
    return false;
});
</script>
{% endblock %}
```

Lorsque le formulaire de recherche est soumis, alors la valeur du champ mot-clé est récupérée puis envoyée par la méthode Ajax à la route **myapp_acteur_rechercher**. Les résultats sont ensuite affichés dans la zone identifiée comme "resultats_recherche". Nous avons également intégré une image d'attente qui s'affichera lors du rechargement de la liste de résultats. Cette image est appelée par le style CSS **.loading** qui fait référence



à **loading.gif**, un fichier téléchargeable ici et à copier dans le répertoire **public/images/**. Si nécessaire relancez la commande "**php app/console assets:install web**" pour transférer le fichier CSS et les images dans le répertoire **web/**.

Voilà, si vous effectuez une recherche avec une partie du nom ou du prénom d'un des acteurs enregistrés, seuls les acteurs correspondants apparaîtront dans la liste des résultats.

Liste des acteurs

Mot-clé :

[Rechercher](#)



Reno Jean 31/07/1948 M [Modifier](#) [Supprimer](#)

V - Une interface multilingue

L'application que nous avons développée jusqu'à présent s'adressait uniquement à des utilisateurs francophones : titres des pages, intitulés des champs de formulaire, messages d'erreur... Tous les textes de l'interface ont été rédigés en français. Pourquoi ne pas élargir le public visé et créer une interface en anglais, en espagnol ou bien dans d'autres langues ? Nous allons voir comment Symfony2 gère les interfaces multilingues grâce à son système de fichiers de langue.

Attention, il ne s'agit pas ici d'avoir une application entièrement multilingue (avec par exemple des fiches de films différentes pour chaque langue) ce qui nécessiterait une modification de la base de données (où plutôt des entités), mais simplement de traduire le texte utilisé dans l'interface utilisateur.

V-A - Choix de la langue avec setLocal()

Tout d'abord, l'utilisateur doit pouvoir choisir la langue qu'il souhaite utiliser pour l'application. Nous allons donc ajouter des liens dans l'interface principale :

Ouvrez le template **layout.html.twig** et insérez le code suivant après le menu principal :

```
// ...
<ul id="menu">
    // ...
</ul>
<div id="choisir_langue">Choisir la langue :
    <a href="{{ path('myapp_choisir_langue', {'langue' : 'fr'}) }}">FR</a> |
    <a href="{{ path('myapp_choisir_langue', {'langue' : 'en'}) }}">EN</a>
</div>
// ...
```

Ouvrez le fichier **MyApp/FilmothequeBundle/Resources/config/routing.yml** et ajoutez la route suivante :

```
myapp_choisir_langue:
    path: /choisir-langue/{langue}
    controller: { _controller: MyAppFilmothequeBundle:Default:choisirLangue }
```

Vous pouvez vérifier que le choix des langues est bien disponible sur l'interface :



ACCUEIL FILMS ACTEURS

Choisir la langue : FR | EN

Enfin, ouvrez le fichier **Controller/DefaultController.php** et insérez la fonction **choisirLangueAction()** :

```
public function choisirLangueAction($langue = null)
{
    if($langue != null)
    {
        // On enregistre la langue en session
        $this->container->get('session')->setLocale($langue);
    }

    // on tente de rediriger vers la page d'origine
    $url = $this->container->get('request')->headers->get('referer');
    if(empty($url)) {
        $url = $this->container->get('router')->generate('myapp_accueil');
    }
    return new RedirectResponse($url);
}
```

La langue choisie est conservée dans une variable de session appelée "Locale". Il s'agit d'une variable prédéfinie dans Symfony2 qui est utilisée dès lors que l'internationalisation intervient : format des dates, fuseau horaire, monnaie courante, texte à traduire, etc. Une fois cette variable définie, il ne reste plus qu'à rediriger l'utilisateur vers sa page de provenance ("referer") ou bien vers la page d'accueil.

V-B - Activation de la traduction

Ouvrez le fichier de configuration de Symfony2 : **Symfony/app/config/config.yml** Dans la section "framework", intégrez le paramètre "translator" comme suit :

```
framework:
    #...
    translator: { fallback: fr }
    #...
```


L'option "fallback" permet de définir la langue qui sera utilisée par défaut au cas où aucune langue n'est choisie ou bien si un terme n'est pas encore traduit.

V-C - Création de fichiers de langue

S'il n'existe pas encore, créez le dossier **MyApp/FilmothequeBundle/Resources/translations/**. Ce répertoire centralisera tous les fichiers de langues de notre application.

Dans ce dossier, créez les fichiers **messages.fr.yml** et **messages.en.yml**. Le premier fichier contiendra tous les éléments textuels en français, alors que le second contiendra ces mêmes éléments, mais traduits en anglais. Pour plus de simplicité, chaque élément textuel sera représenté par un identifiant unique :

Fichier messages.fr.yml :	Fichier messages.en.yml :
<pre> acteur: Acteur acteur.nom: Nom acteur.prenom: Prénom acteur.dateNaissance: Date de naissance acteur.sexe: Sexe acteur.liste: Liste des acteurs acteur.ajouter: Ajouter un acteur # ... </pre>	<pre> acteur: Actor acteur.nom: Last name acteur.prenom: First name acteur.dateNaissance: Date of birth acteur.sexe: Gender acteur.liste: Actors list acteur.ajouter: Add an actor # ... </pre>

Ces listes devront bien entendu être complétées avec le texte relatif aux films et aux autres éléments textuels de l'application. Si vous préférez, vous pouvez également utiliser des fichiers au format XML (xliff) ou PHP à la place des fichiers YML (voir la documentation officielle : <http://symfony.com/doc/current/book/translation.html>).

Nous allons maintenant intégrer ces identifiants dans les différentes parties de l'application qui contiennent du texte : template twig, formulaire, messages d'erreur et les contrôleurs.

V-D - Traduction dans les templates twig avec la fonction trans()

Ouvrez tout d'abord le template Acteur/liste.html.twig et remplacez le texte :

```
Liste des acteurs
```

Par :

```
{{ "acteur.liste" | trans }}
```

Twig doit être informé qu'il devra chercher une traduction pour l'identifiant considéré. Cela se fait grâce à la fonction "trans" (pour "translation" qui signifie traduction en anglais).

Rendez-vous sur la liste des acteurs et essayez maintenant de choisir la langue anglaise. Le titre de la page devrait désormais apparaître en anglais. Si besoin, nettoyez le cache avec la commande "php app/console cache:clear". Si vous souhaitez traduire toute l'interface de l'application, il vous faudra bien sûr reprendre tous les autres éléments textuels présents dans les templates pour n'utiliser que les identifiants définis dans les fichiers de langue.

Vous pouvez tout à fait passer des paramètres dans le texte à traduire comme ceci :

Fichier Acteur/liste.html.twig

```
<p>{{ 'acteur.trouve' | trans({'%nombre%': acteurs|length }) }}</p>
```

Fichier message.fr.yml

```
acteur.trouve: %nombre% acteurs ont été trouvés
```

V-E - Traduction des labels des champs de formulaire

Dans les formulaires, les labels que nous avons personnalisés un peu plus tôt peuvent à leur tour être remplacés par les identifiants utilisés dans les fichiers de langue. Par exemple :

Fichier Form/ActeurForm.php

```
// ...
$builder
    ->add('nom', 'text', array('label' => 'acteur.nom'))
    ->add('prenom', 'text', array('label' => 'acteur.prenom'))
// ...
```

V-F - Traduction des messages d'erreur

Les messages d'erreur que nous avons définis au niveau des entités doivent également être traduits et remplacés par des identifiants uniques. Cependant, Symfony2 recherchera la correspondance dans des fichiers intitulés "validators" et non pas dans les fichiers "messages".

Créez le fichier **validators.fr.yml** et insérez le code suivant :

```
erreur.nom.minLength: Le nom doit avoir au moins {{ limit }} caractères
```

Créez maintenant le fichier **validators.en.yml** pour la traduction anglaise :

```
erreur.nom.minLength: Last name should have {{ limit }} characters minimum
```

Puis, ouvrez l'entité **Acteur.php** et modifiez les annotations comme suit :

```
@Assert\MinLength(limit = 3, message = "erreur.nom.minLength")
```

N'oubliez pas de remplacer chaque message d'erreur par son équivalent. Si besoin, nettoyez le cache de Symfony2 avec la commande : **php app/console cache:clear**

V-G - Traduction au niveau des contrôleurs

Selon la logique de Symfony et plus largement des systèmes MVC ("Model View Controller"), les éléments textuels d'une application font partie de la vue ("View"), tout comme le font les images. Ainsi, il n'est pas conseillé d'utiliser du texte directement dans le code PHP d'un contrôleur. Ceci étant, il est parfois difficile de faire autrement et le processus de traduction doit alors être accessible depuis le code PHP. C'est le cas du texte que nous affichons lorsqu'un acteur vient d'être ajouté. Voici comment le traduire :

Fichier ActeurController.php

```
public function editAction($id = null)
{
    //...
    $message = $this->get('translator')->trans('acteur.ajouter.succes');
    //...
}
```

Fichier messages.fr.yml

```
acteur.ajouter.succes: L'acteur a été ajouté avec succès !
```

Il est bien sûr possible de passer des valeurs de variable dans le texte :

Fichier ActeurController.php

```
public function editAction($id = null)
{
    // ...
    $message = $this->container->get('translator')->trans('acteur.ajouter.succes', array(
        '%nom%' => $acteur->getNom(),
        '%prenom%' => $acteur->getPrenom()
    ));
    // ...
}
```

Fichier messages.fr.yml

```
acteur.ajouter.succes: L'acteur %prenom% %nom% a été ajouté avec succès !
```

Voici le résultat après l'ajout d'un nouvel acteur si vous avez choisi la langue EN :

Add an actor

Actor Jean Dujardin successfully added

Last name

First name

Date of birth

Voilà, l'interface de notre application est maintenant entièrement disponible en français et en anglais ! Quelle que soit l'application que vous désirez développer, je vous conseillerais d'utiliser dès le départ les fichiers de langue et le système de traduction de Symfony2. D'une part les fichiers de langue centralisent l'ensemble des éléments textuels ce qui facilite un quelconque changement de terminologie (ex : utilisation du mot "Comédien" plutôt que "Acteur"). Et d'autre part cela vous évitera de reprendre vos fichiers un par un si jamais vous deviez être amené à traduire votre interface.

VI - Utilisateurs et espace sécurisé

Dans la majorité des cas, les applications web sont destinées à plusieurs utilisateurs :

- les visiteurs qui viennent chercher une information ;
- les rédacteurs qui alimentent le site avec de nouvelles données ;
- les administrateurs qui gèrent les utilisateurs et le contenu du site dans son ensemble.

Cette liste n'est bien sûr pas exhaustive, mais elle permet de se rendre compte que les utilisateurs ont des besoins différents et qu'ils ne doivent pas tous avoir accès aux mêmes pages. Par exemple, un visiteur doit pouvoir rechercher et consulter la fiche d'un film, mais ne doit pas être en mesure la modifier. Le rédacteur doit, quant à lui, pouvoir ajouter et modifier des films et des acteurs, mais il ne doit pas être capable de les supprimer. Enfin, l'administrateur doit avoir accès à toutes les données de l'application pour les consulter, les modifier et éventuellement les supprimer.

Tout utilisateur de l'application doit donc dans un premier temps être identifié, ce que nous pouvons faire par l'intermédiaire d'un formulaire de connexion : c'est l'étape d'authentification. Un utilisateur non connecté sera considéré comme un visiteur anonyme. Selon le rôle qui lui est attribué, l'utilisateur aura accès ou non aux pages

de l'application : c'est l'étape d'autorisation. L'application sera ainsi sécurisée et prête à être gérée par plusieurs personnes. Nous allons voir comment mettre en place une gestion des utilisateurs et un espace sécurisé avec Symfony2.

VI-A - La gestion des utilisateurs avec FOSUserBundle

Plutôt que de réinventer la roue, nous allons utiliser un bundle déjà existant : FOSUserBundle. Ce bundle s'appuie sur le système de sécurité interne à Symfony mais propose en outre une gestion avancée des utilisateurs à qui l'on va pouvoir attribuer les droits d'accès. Ce bundle inclut des formulaires de connexion et d'inscription et propose l'envoi de mot de passe par email en cas d'oubli. Il permet également de suivre les utilisateurs et de créer des groupes. Toutes les fonctionnalités sont configurables et certaines peuvent être désactivées si vous n'en avez pas besoin.

Nous allons utiliser ce bundle pour créer un utilisateur, mettre en place un formulaire de connexion et sécuriser l'accès aux pages de notre application Filmothèque. Je vous invite à lire la documentation sur ce bundle si vous souhaitez utiliser ses (nombreuses) autres fonctionnalités. À noter que ce bundle est en cours de développement et que vous pourrez donc rencontrer des bogues. Mais sa version actuelle est suffisamment avancée pour les besoins que nous avons.

VI-A-1 - Installation de FOSUserBundle

Il est tout d'abord nécessaire de télécharger les fichiers du bundle. Deux méthodes sont possibles : l'utilisation du fichier de dépendance de Symfony2, ou bien le téléchargement classique.

Téléchargement par le fichier deps

- Ouvrez le fichier **deps** situé à la racine de Symfony2.
- À la fin du fichier, ajoutez le code suivant :

```
[FOSUserBundle]
git=git://github.com/FriendsOfSymfony/FOSUserBundle.git
target=bundles/FOS/UserBundle
```

- Puis lancez la commande **php bin/vendors install --reinstall**.
- Cette commande va vérifier sur github si les bundles tiers ("vendors") ont été modifiés et les mettra à jour si nécessaire.
- Une fois la mise à jour terminée, vous pouvez vérifier que le FOSUserBundle est bien présent dans le répertoire **Symfony2/vendor/bundles/FOS/UserBundle/**.

Téléchargement classique

- Téléchargez la dernière version de FOSUserBundle à l'adresse suivante : <https://github.com/FriendsOfSymfony/FOSUserBundle>.
- Décompressez le contenu du fichier dans le répertoire (à créer si nécessaire) : **Symfony2/vendor/bundles/FOS/UserBundle/**.

Il faut ensuite informer Symfony2 de la présence de ce nouveau bundle.

Ouvrez le fichier **app/AppKernel.php** et insérez la ligne :

```
public function registerBundles()
{
    $bundles = array(
        // ...
        new FOS\UserBundle\FOSUserBundle(),
    );
}
```

Ouvrez le fichier `Symfony2/app/autoload.php` et insérez la ligne :

```
$loader->registerNamespaces(array(
    // ...
    'FOS' => __DIR__.'../vendor/bundles',
));
```

Ce bundle est désormais installé et prêt à être utilisé. Comme son code a été développé et sera maintenu par d'autres développeurs, nous l'avons placé dans le répertoire "vendors" qui contient toutes les applications tierces susceptibles d'être utilisées. Il n'est pas conseillé de placer un bundle tiers directement dans votre application (répertoire src/). En effet si une nouvelle version du bundle est disponible et que vous souhaitez l'intégrer, cela peut avoir des répercussions sur votre application qui risque de ne plus fonctionner correctement.

VI-A-2 - Utiliser FOSUserBundle au sein de l'application

Pour utiliser FOSUserBundle, nous allons créer dans notre application un nouveau bundle qui sera défini comme une extension de FOSUserBundle. Nous pourrions ainsi réaliser des modifications directement dans le code de notre bundle plutôt que de toucher au code original de FOSUserBundle.

Ouvrez le fichier `app/config/config.yml` et insérez à la fin les lignes suivantes :

```
fos_user:
    db_driver:      orm
    firewall_name:  main
    use_listener:   false
    user_class:     MyApp\UtilisateurBundle\Entity\Utilisateur
```

Puis, créez le nouveau bundle "**MyApp/UtilisateurBundle**" grâce à la commande `php app/console generate:bundle`.

Si besoin, enregistrez-le dans Symfony2 en modifiant le fichier `app/AppKernel.php` :

```
public function registerBundles()
{
    $bundles = array(
        // ...
        new MyApp\UtilisateurBundle\MyAppUtilisateurBundle(),
        // ...
    );
}
```

Ouvrez ensuite le fichier `MyApp/UtilisateurBundle/MyAppUtilisateurBundle.php` et insérez le code suivant :

```
// ...
class MyAppUtilisateurBundle extends Bundle
{
    public function getParent()
    {
        return 'FOSUserBundle';
    }
}
```

Créez l'entité `Utilisateur.php` dans le répertoire `MyApp/UtilisateurBundle/Entity/` :

```
<?php
namespace MyApp\UtilisateurBundle\Entity;
```

```
use FOS\UserBundle\Entity\User as BaseUser;
use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity
 */
class Utilisateur extends BaseUser
{
    /**
     * @ORM\Id
     * @ORM\Column(type="integer")
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    protected $id;

    public function __construct()
    {
        parent::__construct();
    }
}
```

Cette entité hérite de BaseUser qui est l'utilisateur par défaut de FOSUserBundle.

Il faut maintenant générer les entités : **php app/console doctrine:generate:entities MyApp**

Et mettre à jour le schéma de la base de données : **php app/console doctrine:schema:update --force**

VI-A-3 - Créer un premier utilisateur

FOSUserBundle dispose de plusieurs commandes qui permettent de gérer les utilisateurs directement depuis la console. Nous allons nous en servir pour créer un premier utilisateur.

Ouvrez votre console et tapez la ligne suivante : **php app/console fos:user:create username email password**

Bien entendu il vous faut remplacer "username", "email" et "password" par le nom d'utilisateur (ex : admin), l'adresse email (ex : contact@filmotheque.com) et le mot de passe de votre choix.

Si vous souhaitez définir ce nouvel utilisateur comme administrateur principal, utilisez la commande : **php app/console fos:user:promote username** (remplacez "username" par le nom d'utilisateur que vous venez de créer) et choisissez le rôle **ROLE_ADMIN**.

Vous avez désormais un nouvel utilisateur qui sera considéré comme administrateur.

VI-A-4 - Intégration des routes et des templates de FOSUserBundle

Notre base de données a été mise à jour et a désormais la possibilité de gérer des utilisateurs. Cependant, il nous faut continuer l'intégration de FOSUserBundle pour que ses pages soient accessibles depuis notre application.

Commençons par définir les routes. Nous pourrions définir nos propres routes mais par simplicité nous allons importer celles déjà définies FOSUserBundle.

Ouvrez le fichier **Symfony2/src/MyApp/FilmothequeBundle/config/routing.yml** et ajoutez :

```
fos_user_security:
    resource: "@FOSUserBundle/Resources/config/routing/security.xml"
    prefix: /

fos_user_profile:
```

```
resource: "@FOSUserBundle/Resources/config/routing/profile.xml"
prefix: /profile

fos_user_register:
resource: "@FOSUserBundle/Resources/config/routing/registration.xml"
prefix: /register

fos_user_resetting:
resource: "@FOSUserBundle/Resources/config/routing/resetting.xml"
prefix: /resetting

fos_user_change_password:
resource: "@FOSUserBundle/Resources/config/routing/change_password.xml"
prefix: /change-password
```

Modifions maintenant le template par défaut de FOSUserBundle pour qu'il hérite du template principal layout de notre Filmothèque.

Dans le répertoire **MyApp/UtilisateurBundle/Resources/views**, créez le fichier **layout.html.twig** et insérez le code :

```
{% extends "MyAppFilmothequeBundle::layout.html.twig" %}

{% block title %}{% endblock %}

{% block content %}
    {% block fos_user_content %}{% endblock %}
{% endblock %}
```

Placé dans le bundle Utilisateur, ce template va remplacer le fichier layout par défaut de FOSUserBundle. Cela est possible car UtilisateurBundle hérite de FOSUserBundle. Et comme ce nouveau layout hérite du layout de notre bundle Filmothèque, et bien lorsque nous accéderons à une page de FOSUserBundle, elle sera intégrée à la mise en page de notre application.

Essayez maintenant d'accéder au formulaire d'inscription en vous rendant sur : http://localhost/Symfony2/web/app_dev.php/myapp/register/

Vous devriez voir apparaître le formulaire intégré dans l'interface de notre application.

VI-B - Gestion des droits d'accès

VI-B-1 - Activer la sécurité

Le framework Symfony2 considère la sécurité d'une application un peu comme le pare-feu d'un ordinateur. Toute demande d'accès à une page passera d'abord par le pare-feu qui vérifiera que l'utilisateur est bien identifié. Puis, le système vérifiera que cet utilisateur possède bien les droits d'accès à la page demandée. Les notions d'utilisateur, de firewall et de contrôle d'accès font partie intégrante du système de sécurité de Symfony2. Le bundle FOSUserBundle s'appuie sur ce système pour fournir une gestion améliorée des utilisateurs et de la sécurité. Nous allons configurer Symfony2 et l'informer de la provenance des utilisateurs ("providers") et des options du pare-feu ("firewall"). Cela passe par le fichier **security.yml** de Symfony2.

Ouvrez le fichier **Symfony2/app/config/security.yml** et remplacez son contenu par :

```
imports:
- { resource: "@MyAppFilmothequeBundle/Resources/config/security.yml" }
```

Puis créez le fichier **security.yml** dans le répertoire **MyApp/FilmothequeBundle/Resources/config/** et insérez le code suivant :

```
security:
  providers:
    fos_userbundle:
      id: fos_user.user_manager

  firewalls:
    main:
      pattern: .*
      form_login:
        provider: fos_userbundle
        login_path: /myapp/login
        use_forward: false
        check_path: /myapp/login_check
        failure_path: null
        default_target_path: /myapp
      logout:
        path: /myapp/logout
        target: /myapp
      anonymous: true
```

Nous avons ici précisé que les utilisateurs sont gérés par FOSUserBundle. Nous avons également indiqué quels sont les accès au formulaire de connexion, ainsi qu'aux pages de vérification et de déconnexion de l'utilisateur.

VI-B-2 - Accès au formulaire de connexion

Sur l'ensemble des pages de notre application, nous allons afficher un lien vers le formulaire de connexion. Une fois l'utilisateur connecté, nous afficherons un lien pour se déconnecter.

Dans le répertoire **MyApp/UtilisateurBundle/Resources/views**, créez le fichier **connexion.html.twig** et insérez le code :

```
<div>
{% if is_granted("IS_AUTHENTICATED_REMEMBERED") %}
  <a href="{{ path('fos_user_profile_show') }}">{{ 'layout.logged_in_as'|trans({'%username%':
app.user.username}, 'FOSUserBundle') }}</a> |
  <a href="{{ path('fos_user_security_logout') }}">
    {{ 'layout.logout'|trans([], 'FOSUserBundle') }}
  </a>
{% else %}
  <a href="{{ path('fos_user_security_login') }}">{{ 'layout.login'|trans([], 'FOSUserBundle') }}</a>
  |
  <a href="{{ path('fos_user_registration_register') }}">{{ 'layout.register'|trans([],
'FOSUserBundle') }}</a>
{% endif %}
</div>

{% for key, flash in app.session.getFlashes() %}
<div class="{{ flash }}">
  {{ key|trans([], 'FOSUserBundle') }}
</div>
{% endfor %}
```

Puis, dans le fichier **MyApp/Filmotheque/Resources/views/layout.html.twig**, après le choix de la langue, incluez le template contenant le lien connexion :

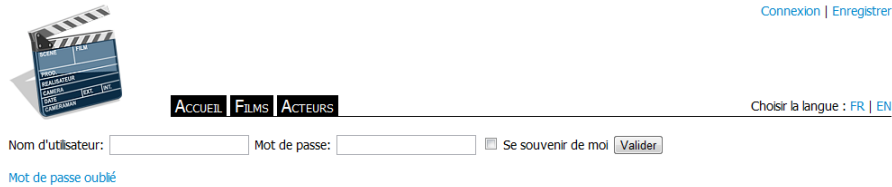
```
<ul id="menu">
  // ...
</ul>

<div id="choisir_langue">Choisir la langue :
  // ...
</div>
```



```
<div id="connexion">
    {% include "MyAppUtilisateurBundle::connexion.html.twig" %}
</div>
```

Rendez-vous sur l'application Filmothèque puis cliquez sur le lien "Connexion" qui a dû apparaître en haut à droite. Le formulaire de connexion devrait s'afficher :



Vous pouvez si besoin améliorer la mise en page du formulaire de connexion. Pour cela, il ne faudra pas toucher aux fichiers FOSUserBundle mais continuer à profiter de l'héritage mis en place avec notre bundle Utilisateur. Il vous faudra copier/coller le fichier **views/Security/login.html.twig** du bundle FOSUserBundle et vers le répertoire **MyApp/UtilisateurBundle/Resources/views/Security** avant d'effectuer les modifications souhaitées. De la même façon, vous pourrez améliorer la présentation de la liste des utilisateurs.

VI-B-3 - Configurer les pages aux accès restreints

Le formulaire de connexion est en place, mais pour l'instant l'ensemble de nos pages restent accessibles à tout le monde. Voyons comment définir les pages qui auront un accès restreint.

À la fin du fichier **MyApp/FilmothequeBundle/Resources/config/security.yml**, ajoutez :

```
security:
    // ...

    role_hierarchy:
        ROLE_ADMIN:      ROLE_USER
        ROLE_SUPERADMIN: ROLE_ADMIN

    access_control:
        # Liste des pages accessibles à tous les utilisateurs (ne pas toucher)
        - { path: ^/_wdt/, role: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/_profiler/, role: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/_js/, role: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/_css/, role: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/login$, role: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/login_check$, role: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/register$, role: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/resetting$, role: IS_AUTHENTICATED_ANONYMOUSLY }

        # PAGES ACCESSIBLES AUX ADMINISTRATEURS
        - { path: ^/myapp/admin/, role: ROLE_ADMIN }

        # PAGES ACCESSIBLES AUX UTILISATEURS CONNECTES
        - { path: ^/myapp/change-password, role: ROLE_USER }

        # PAGES ACCESSIBLES À TOUS
        - { path: ^/myapp.*, role: IS_AUTHENTICATED_ANONYMOUSLY }
```

Le contrôle d'accès aux pages est défini par le paramètre "access_control" qui permet d'associer des pages, ou plutôt des URL, aux différents rôles prédéfinis (USER, ADMIN...). Ainsi, les formulaires de connexion et d'inscription ainsi que les fichiers JavaScript et CSS seront accessibles quel que soit le rôle, sans quoi un utilisateur ne pourrait pas s'identifier. De même, la page d'accueil de notre application pourra être visible par tout le monde. En revanche, seuls les utilisateurs connectés (USER) pourront accéder au formulaire de changement de mot de passe (/myapp/change-password). Enfin, toutes les URL qui commenceront par **myapp/admin/** seront protégées et accessibles uniquement par les administrateurs.

Pour que la protection des pages de notre application fonctionne, il est maintenant nécessaire de modifier les routes des différents formulaires et des liens de suppression. Il suffit pour cela d'ajouter le préfixe "admin/" aux URL concernées :

Fichier MyApp/FilmothequeBundle/Resources/config/routing.yml

```
myapp_acteurajouter:
  pattern: /admin/acteur/ajouter
  defaults: { _controller: MyAppFilmothequeBundle:Acteur:editier }
myapp_acteurmodifier:
  pattern: /admin/acteur/modifier/{id}
  defaults: { _controller: MyAppFilmothequeBundle:Acteur:editier }
myapp_actorsupprimer:
  pattern: /admin/acteur/supprimer/{id}
  defaults: { _controller: MyAppFilmothequeBundle:Acteur:supprimer }
// ...
```

Si vous essayez maintenant d'accéder à la page d'ajout d'acteur, vous serez automatiquement redirigé vers le formulaire de connexion. Si vous vous connectez, vous arriverez sur le formulaire d'ajout d'acteur et verrez votre nom d'utilisateur et le lien de déconnexion s'afficher :



Connecté en tant que admin | Déconnexion

ACCUEIL FILMS ACTEURS

Choisir la langue : FR | EN

Les pages sont bien protégées, mais les liens vers les actions d'ajout, de modification et de suppression sont toujours visibles quel que soit l'utilisateur. Nous pouvons mettre en place une condition à l'affichage de ces liens d'administration :

Acteur/lister.html.twig

```
// ...
{% if is_granted('ROLE_ADMIN') %}
  <p><a href="{{ path('myapp_acteurajouter') }}">Ajouter un acteur</a><p>
{% endif %}
// ...
```

Ici, seuls les utilisateurs ayant le rôle "ROLE_ADMIN" verront le lien d'ajout d'un acteur. Vous pouvez mettre en place ces conditions sur l'ensemble des liens d'ajout, de modification et de suppression de l'application.

VII - Conclusion

Au cours de ce tutoriel nous avons présenté de nombreux aspects du framework Symfony2 tout en développant une application complète de gestion de films et d'acteurs. Nous avons profité de l'héritage des templates twig et de l'outil assets pour améliorer l'interface graphique et la mise en page de notre application. Nous avons également modifié les options des formulaires pour les personnaliser selon nos besoins. L'intégration de JQuery a simplifié l'accès à la méthode Ajax pour un formulaire de recherche dynamique, alors que la création de fichiers de langue nous a permis de mettre en place une interface en français et en anglais. Enfin, nous avons abordé les aspects de sécurité à travers le bundle FOSUserBundle qui nous a permis de créer des utilisateurs et de contrôler l'accès aux pages d'administration.

Rassurez-vous, il vous reste bien d'autres subtilités de Symfony2 à découvrir : l'upload de fichiers, l'envoi d'emails, les formulaires imbriqués, les tests utilisateurs, etc. Je vous renvoie vers la documentation officielle de Symfony2 et vers votre moteur de recherche préféré...

La rédaction de ce tutoriel (et du précédent) n'aurait pu se faire sans le travail considérable effectué par les développeurs du framework Symfony2 que je tiens à remercier ici. Je crois sincèrement que la deuxième version

entièrement repensée de ce framework vous permettra de développer des applications web de qualité, sécurisées, faciles à maintenir, et ce en un minimum de temps.

Une dernière remarque, pour faciliter la compréhension de ce tutoriel nous avons francisé la plupart des termes utilisés. Nous avons par exemple créé l'entité "Acteurs" et la fonction "Ajouter". Ce n'est pas une pratique courante. Si vous souhaitez que votre code puisse être facilement réutilisé, préférez des termes anglais à tous les niveaux : entités, variables, méthodes, nom de fichiers, identifiants des fichiers de langues, etc.

Pour finir, il ne me reste plus qu'à vous proposer de télécharger l'application Filmotheque finalisée :
[Source](#) [MyApp.zip](#)

VIII - Références

- **Symfony** : Site officiel de Symfony2 [www.symfony.com](#)
- **Twig** : Site officiel de Twig [www.twig-project.org](#)
- **Doctrine** : Site officiel de Doctrine [www.doctrine-project.org](#)
- **FriendsOfSymfony** : groupe de développeurs qui réalisent des bundles pour Symfony2, bundles que vous pourrez réutiliser dans vos applications <http://friendsofsymfony.github.com/>

IX - Remerciements

Je tiens à remercier [dourouc05](#) et [djamalo](#) pour leurs conseils lors de la rédaction de ce tutoriel ainsi que [jpcheck](#) et [ClaudeLELOUP](#) pour leur relecture.