* Le projet BMG avec SYMFONY

# Présentation

## Description du thème

|  |  |
| --- | --- |
| Propriétés | Description |
| **Intitulé long** | Gestion des utilisateurs, des ouvrages et des prêts de la bibliothèque de Groville en utilisant le framework Symfony 3 |
| **Formation concernée** | BTS SIO SLAM 4 |
| **Présentation** | Le support propose de réécrire entièrement le projet Bibliothèque municipale de Groville (version 4 – MVC) en utilisant le framework Symfony 3, le moteur de template Twig et l'ORM Doctrine. |
| **Notions** | Modèle MVC, moteur de template de pages, ORM, formulaires Symfony. |
| **Pré-requis** | * PHP, MVC, php objet, framework Symfony, Doctrine * mission 4 du projet BMG : le code s'appuie sur les fichiers de ce projet. |
| **Outils** | Framework Symfony 3, environnement de développement NetBeans |
| **Mots-clés** | Modèle MVC, template, Framework, Symfony 3, ORM |
| **Durée** | 6 heures |
| **Auteur(es)** | Projet BMG MVC : Darius Kugel – réécriture en Symfony 3  : Catherine Lebeau |

**Présentation**

L’application BMG (mission 4) version MVC est ici re-développée en utilisant Symfony 3.

Il ne s’agit pas d’un cours sur Symfony ; plusieurs sites font une présentation détaillée de découverte du produit largement suffisante pour débuter. Deux sites en particulier sont à consulter :

* <https://symfony.com/doc/current/index.html> : le site officiel (en anglais)
* https://openclassrooms.com/courses/developpez-vos-applications-web-avec-symfony2 : un support très bien fait sur *OpenClassroom de* ***Symfony.***

**Les choix**

Symfony propose par défaut une architecture MVC.

La couche modèle de l'application originale reposait sur la classe PdoDao pour l’accès aux données. Nous utiliserons dans ce TP, le moteur ORM Doctrine ainsi que la gestion des formulaires par Symfony.

Nous utiliserons également le moteur de template *twig* pour les vues ; ce n'est pas obligatoire, il est possible d'utiliser le PHP.

Ainsi qu'il est précisé dans les prérequis, le code s'appuie sur le code de la mission 4 du projet BMG. Vous devez donc avoir sous la main tous les fichiers de ce projet, que nous modifierons pour les adapter à Symfony, Doctrine et Twig.

# Installation de l'application

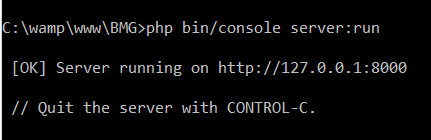
## Installation

Je vous demanderai dans ce tutoriel de respecter EXACTEMENT les noms de fichiers, dossiers, emplacements **y compris la casse**, ceci afin de faciliter le débogage.

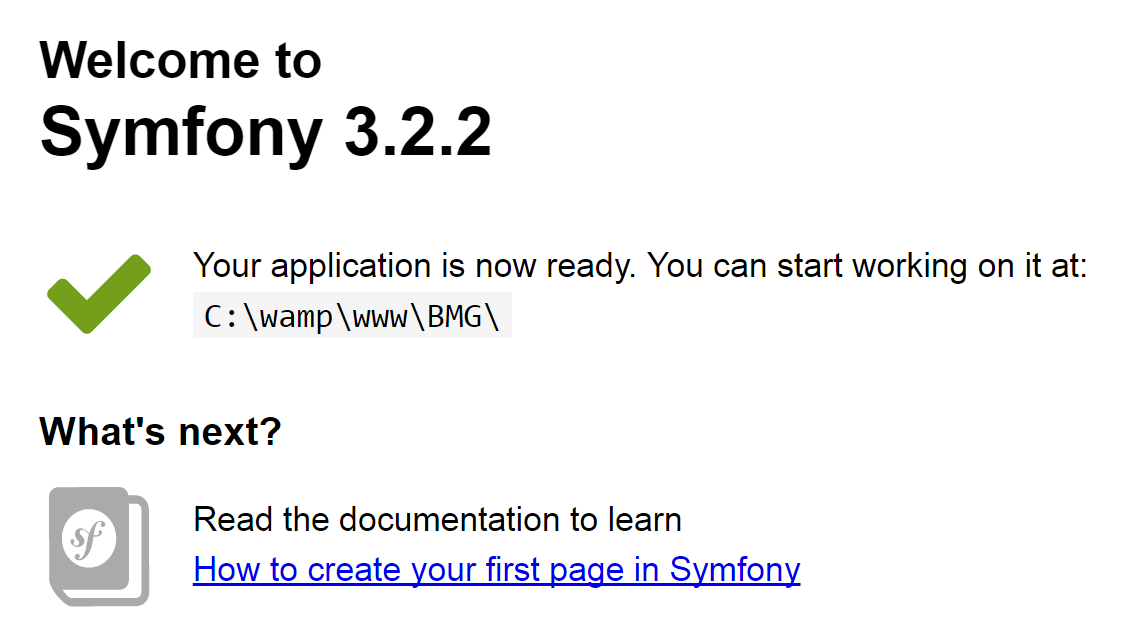
* Créer une nouvelle application Symfony BMG : .

(ou récupérer le fichier BMGSymfony.zip)

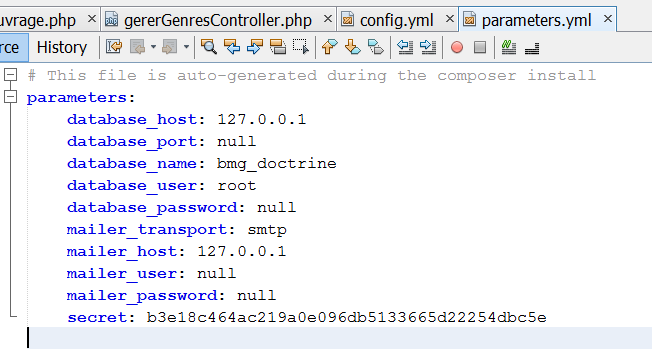
* Démarrer le serveur en ligne de commande :



* Tester en tapant l'URL dans le navigateur " localhost:8000" et on obtient :



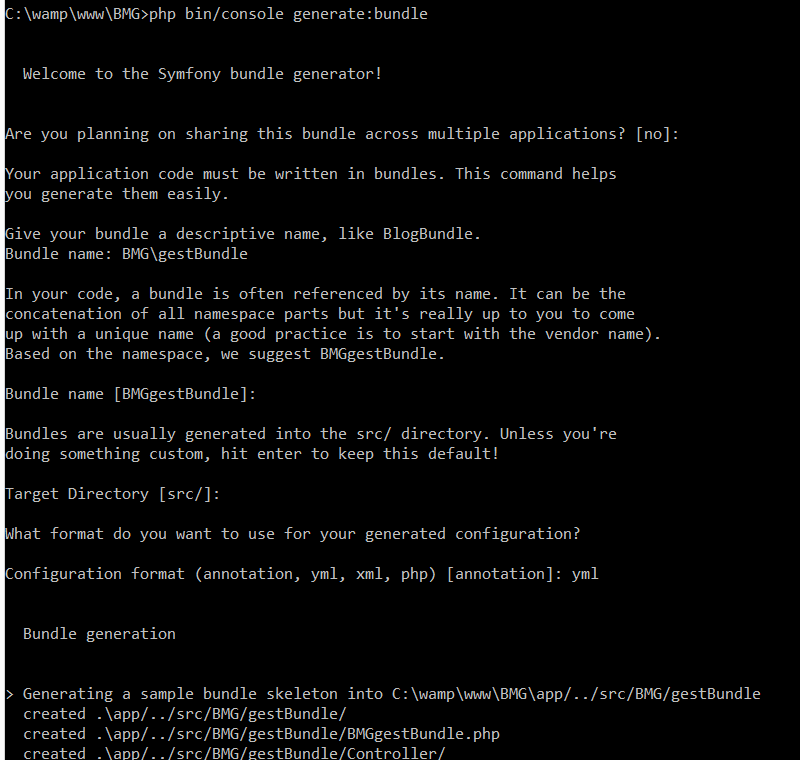
* Afin de ne pas interférer avec le travail effectué en SLAM3, nous travaillerons sur une base indépendante : bmg\_doctrine 🡪 copier la base bmg existante en une base équivalente nommée bmg\_doctrine.
* Mettre à jour le nom de la base de données utilisée par notre application Symfony, dans le fichier *parameters.yml* (ici database\_name : bmg\_doctrine) :



## Création du Bundle

Nous allons créer le Bundle de gestion de la bibliothèque. Ce bundle s'appellera : BMGgestBundle.

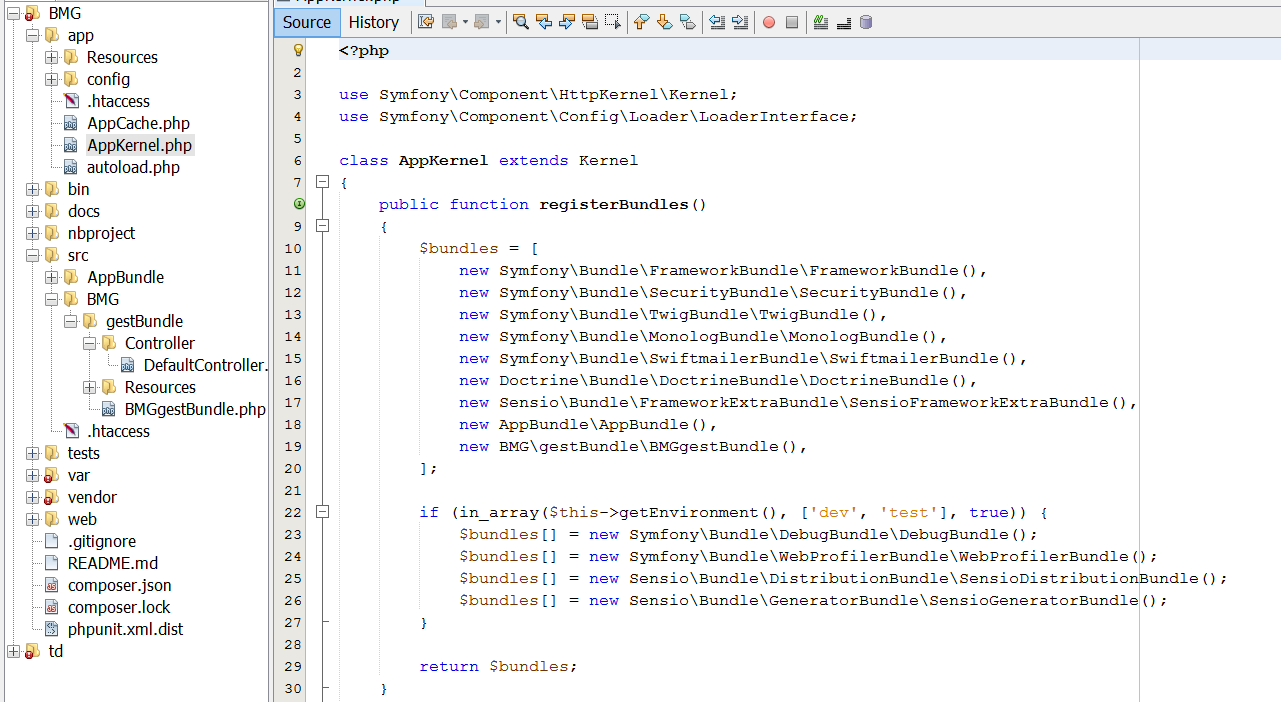
**Remarque** : Attention au nom que vous donnez à votre Bundle dans la fenêtre de commande : BMG\gestBundle.



* Créer le projet dans Netbeans (Nouveau projet, php application with existing sources).

On peut observer le code que Symfony a créé lors de la création du Bundle :

* Un répertoire BMG/gestBundle,
* L'instanciation d'un Bundle nommé BMGgestBundle dans appKernel.php.



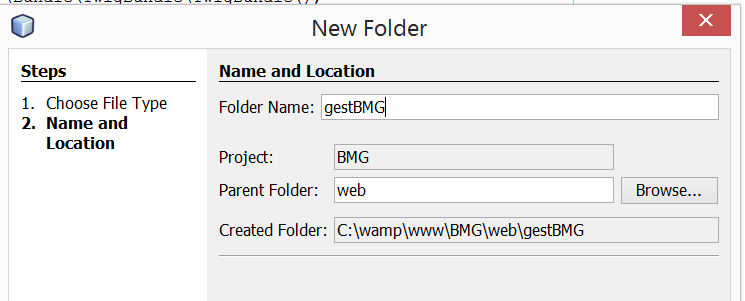
## La page d'accueil du site BMG

### Préparation

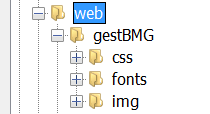
Nous allons commencer par intégrer des éléments de l’application *BMG*.

Dans un premier temps, concentrons-nous sur l’unique page d’accueil. Copions les trois seuls types de ressources **publics** du site BMG, les images, les polices et les fichiers CSS. Toutes les autres ressources, y compris le code, sont privées dans Symfony.

* Au préalable, créons un répertoire *gestBMG* dans le dossier *web* :



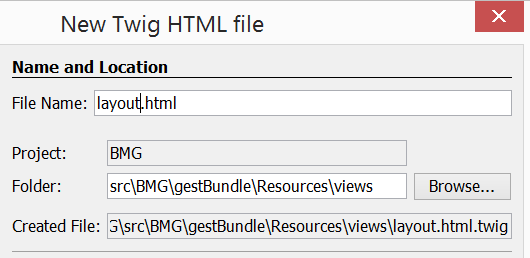
* Puis, copions les trois répertoires de BMG (css, fonts et img) dans ce répertoire public web/gestBMG.



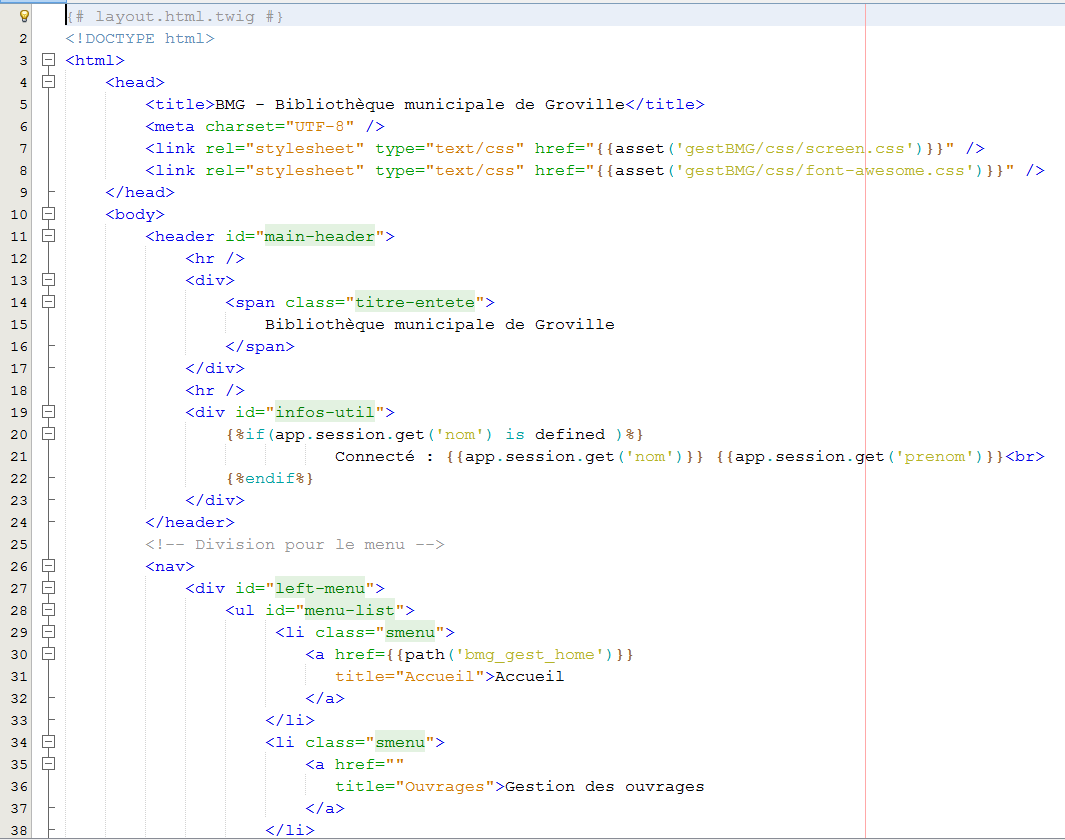
### La vue index.html.twig

Toutes les vues de l'application BMG ont la même structure : elles contiennent l'entête, le menu, un contenu et un pied de page. Seul le contenu diffère.

Créer **à la racine du dossier BMG/gestBundle/Resources/views**, une vue de type html.twig nommée layout.html.twig. Cette vue de base résumera la structure de toutes les pages du bundle.Toute les autres vues hériteront de cette vue.



La vue layout.html.twig est la suivante :



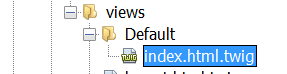


Elle regroupe les codes des vues *v\_header.php, v\_menu.php, v\_footer.php*.

**Remarques** :

* Les lignes 7 et 8 sont modifiées : elles utilisent le langage *twig* et sa fonction ***asset*** qui permet de ne pas définir des URI "en dur".
* La gestion des sessions avec Symfony : lignes 20 et 21. Assez simples à interpréter, elles permettent de récupérer le contenu de la variable de session : similaire à *$\_SESSION[nom’].*
* Nous prévoyons des blocs qui seront surchargés dans les vues qui vont hériter du *layout* (58 à 61). Ces blocs sont nommés "contenu" et "contenuSuite"
* La double accolade est interprétée par *twig* comme "met le contenu", c’est l’équivalent du < ?php echo $quelquechose ?>
* Les balises {% %} sont interprétées comme "fait ce qui est défini", ici c’est de définir simplement des blocs (ici lignes 58 à 61).
* Les URL pointées dans le menu ont été mises à la sauce "Symfony" en utilisant la fonction "path()": ligne 30, une seule route nommée 'bmg\_gest\_home' est définie pour le moment. On listera les routes correspondantes dans le fichier *routing*.

|  |  |
| --- | --- |
|  |  |

**La page d'accueil : index.html.twig** 

Copions le code situé dans le <div = "contenu"> de la page *v\_home.php* dans la page créée par défaut du répertoire par défaut (Default) *index.html.twig* ; ceci doit donner le code décrit ci-dessous :

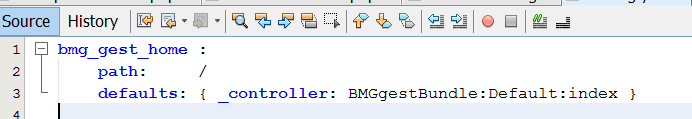


**Remarques :**

* Le bloc *extends* indique l’héritage (ligne 1).
* Comme dans la vue précédente, l'utilisation des *asset* pour définir les attributs *src* des balises *img*.
* Notez la syntaxe du nommage du fichier père (*layout*), les deux points (::) indique **l’absence** du répertoire puisque le fichier est à la racine de l'application dans un dossier *views*.

Rappelez-vous : à la création du bundle, Symfony a créé une route dans *routing.yml* et un contrôleur (*DefaultController.php*) par défaut.

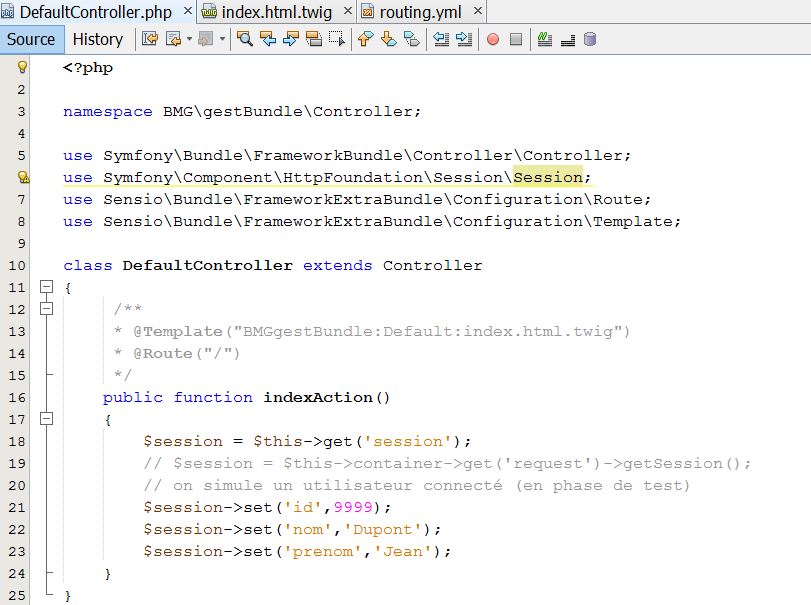
Renommer la route **créée par défaut et modifier ainsi :**



**Remarques :**

Le code de la fonction indexAction(), située dans le contrôleur DefaultController.php est exécuté à l'arrivée dans l'application.

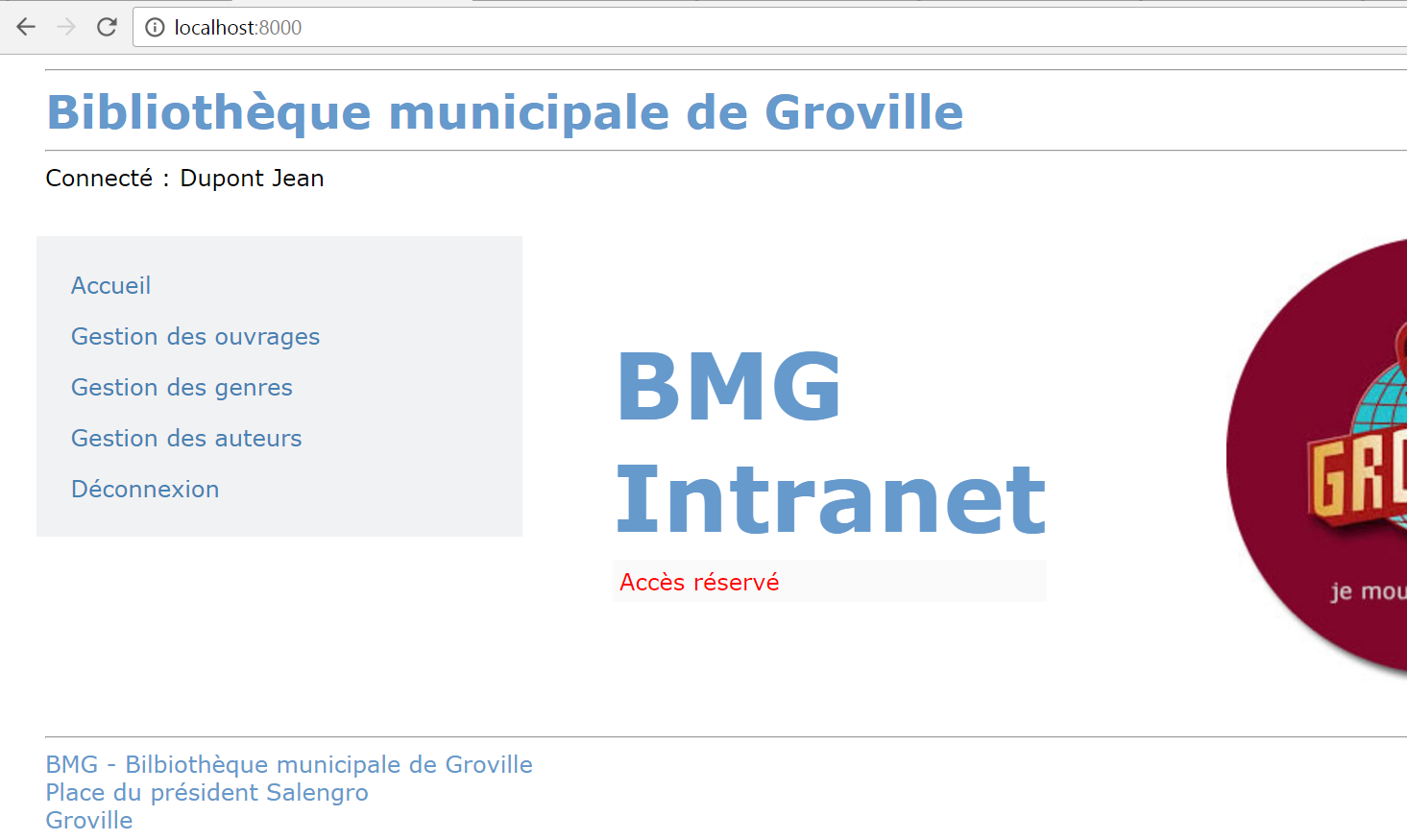
Modifions également le contrôleur afin qu'il appelle la vue créée précédemment.



**Remarques :**

* L'annotation permet d'appeler la vue : on évite le "render"
* La ligne 18 récupère la session courante ; à noter que le contrôleur (*this*) y a directement accès.
* Les lignes 21 et suivantes enregistrent en session les données du visiteur.

Il ne reste plus qu'à tester : l'URL "localhost:8000" ouvre la page d'accueil.



# Doctrine

Pour développer les pages suivantes, nous aurons besoin d'affiner le modèle, plus spécifiquement le lien avec la base de données BMG. Nous aurons recours à l'ORM Doctrine.

Grâce à Doctrine, nous pourrons "mapper" la base de données existante et créer automatiquement les classes métier correspondantes aux tables. Nous accèderons à ces tables via les objets créés.

Les étapes de création des entités Doctrine sont expliquées dans le fichier <RétroconceptionBMG.docx>.

# La gestion des genres

Comme dans votre projet BMG MVC, la gestion des genres va inclure différentes actions :

* consulterGenre ($id)
* ajouterGenre
* modifierGenre ($id)
* supprimerGenre ($id)
* listerGenres

Chacun de ces cinq cas sera traité dans le contrôleur gererGenresController.php via une fonction Action().

Bien sûr, une route sera créée permettant l'aiguillage vers chacune de ces actions.

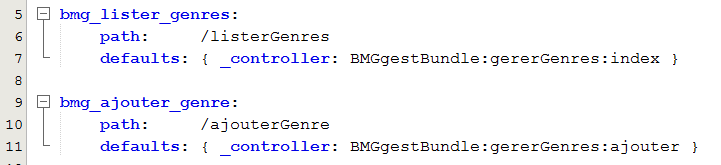
Dans chacun des cas traités, nous procéderons en trois temps :

* création d'une route (dans src/BMG/gestBundle/Resources/config/routing.yml),
* modification du contrôleur gererGenresController.php (ajout d'une fonction Action() )
* puis création d'une vue.

## Lister les genres

### La route (fichier routing.yml)

Nous allons commencer par ajouter deux nouvelles routes :



### Le contrôleur

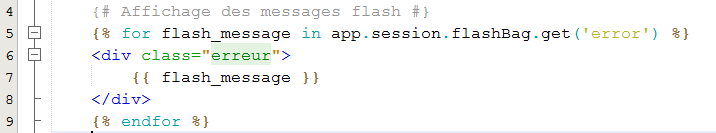
On voit dans la route bmg\_lister\_genres que nous allons créer la fonction indexAction() dans le contrôleur *gererGenresController.php*.

* Créer le contrôleur "*gererGenresController.php* " dans le dossier Controller.
* La fonction indexAction() va effectuer les deux mêmes actions que l'action listerGenres de l'application originale :
  1. Récupération de la liste des genres de la base de données dans un tableau d'objets Genre
     + Récupération de l'Entity Manager via l'objet contrôleur ($this) : variable $em,
     + Récupération du repository lié à l'entité Genre via l'Entity Manager : variable $repository\_genre,
     + La méthode findAll() appelée par le repository récupère **toutes** les lignes de la table Genre pour les placer dans un tableau d'objets Genre : variable $listeGenres.
  2. Appel de la vue en passant cet objet en paramètre .



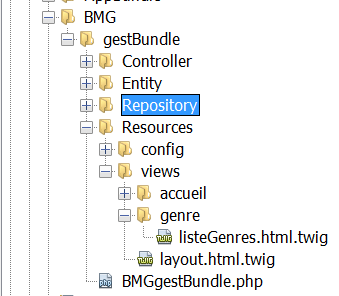
**Remarque :**

* Ligne 8 : ajout du code de l'entité Genre au contrôleur,
* Ligne 14 : La recherche des genres via Doctrine se fait dans un "try",
* Ligne 24 : Le "catch" crée un message flash d'erreur. Il est suivi d'une redirection sur la page d'accueil. Le message sera affiché en ajoutant dans la page d'accueil /*Default/index.html.twig* le code suivant :

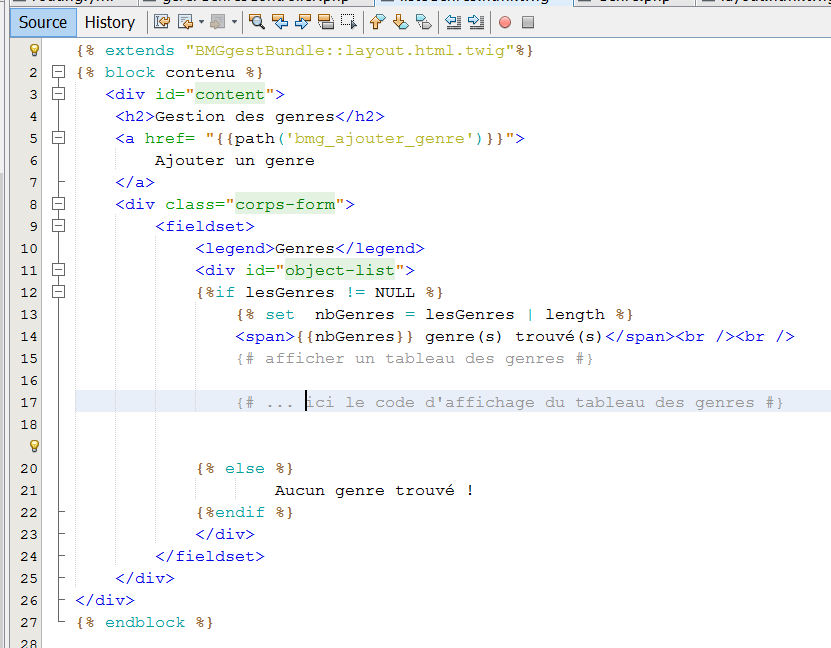


### La vue

* Créer un dossier *genre* dans le dossier *views* puis une vue *listeGenres.html.twig*



* Copier le contenu de la vue v\_listeGenres.php dans la vue listeGenres.html.twig
* Nous procèderons en plusieurs étapes qui seront testées à chaque fois. Dans un premier temps mettez en commentaire l'affichage du tableau ( {# Commentaire #} )
* Modifier le code de cette façon :



**Remarques :**

* Là encore, cette vue hérite du layout (ligne 1), le bloc "contenu" est surchargé (ligne 2).
* Ligne 5 : notez la syntaxe de traitement du lien "ajouter un genre" : la fonction *path()* s’occupe des répertoires (c’est bien pratique) et on se concentre sur l’essentiel : quelle route prendre ? C’est-à-dire quelle URL et quel code exécuter, c’est justement indiqué dans la nouvelle route ajoutée dans le paragraphe précédent. Cette fonctionnalité sera développée ultérieurement.
* Ligne 12 : la vue *twig* utilise le paramètre passé par le contrôleur via le render : lesGenres
* Observez le code *twig* : le if … else … endif , l'affectation d'une variable (set), la récupération de la longueur du tableau lesGenres (| length ) …

|  |  |
| --- | --- |
| Testons (localhost:8000/listerGenres ou en cliquant sur le lien du menu après l'avoir codé) ; nous obtenons : |  |

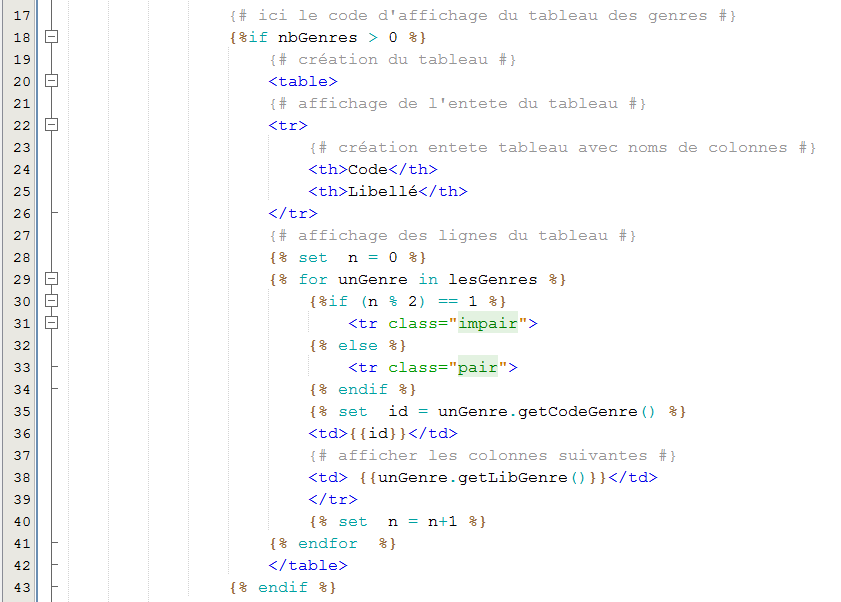
Si cela fonctionne pas, voilà quelques pistes :

* Erreur dans la route ?
* Erreur dans le code du contrôleur ?
* Erreur dans la vue Twig ?
* Oubli de mettre à jour le nom de la base de données (paramètres symfony) ?

Ajoutons à présent l'affichage du tableau dans la vue *Twig*.

Dans cette deuxième étape, nous afficherons d'abord le tableau sans coder le lien sur l'ID, qui, je vous le rappelle, permet de consulter un genre.

Code à insérer après la ligne 15 :



**Remarque :**

* Twig : On observera le parcours de la collection (for … endfor) et la notation objet (unGenre**.**getCodeGenre()) dans Twig,
* Ligne 36 : {{ variable }} : permet d'afficher le contenu d'une variable,
* Doctrine : unGenre est un objet de la classe Genre créé automatiquement par Doctrine. On utilise également les accesseurs que Doctrine a créé au cours du paragraphe précédent .

Tester. 

## Consulter un genre

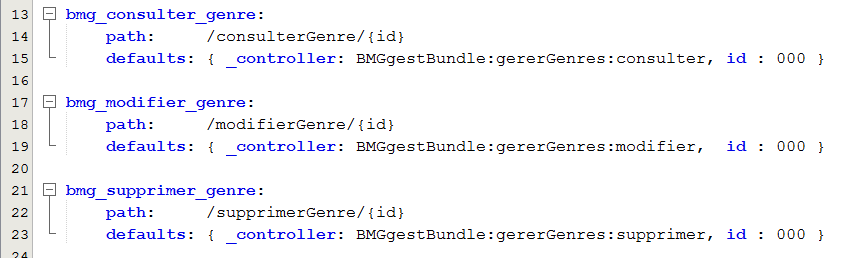
Comme dans la mission 4 BMG, le clic sur le code du genre dans la vue précédente listeGenres.html.twig, permet de consulter un genre et d'en afficher les informations. Sont également présents sur cette vue les liens suivants :

* Modifier le genre,
* Supprimer le genre,
* Ajouter un genre .

Donc autant de routes, contrôleur et vues à coder. Nous préparerons donc les fichiers et les fonctions.

### Les routes

Ajouter les routes suivantes dans le fichier *routing.yml.*



**Remarques** :

Pour chacune des trois routes, un paramètre 'id' sera passé dans l'URL (et récupéré par la fonction correspondante du contrôleur) : {id}. Si ce paramètre est omis (il s'agit d'un cas d'erreur), il prendra la valeur 000.

### L'action consulterAction() dans le contrôleur

Après lecture des routes ci-dessus, vous aurez compris que dans **gererGenresController.php** , nous aurons de nouvelles fonctions :

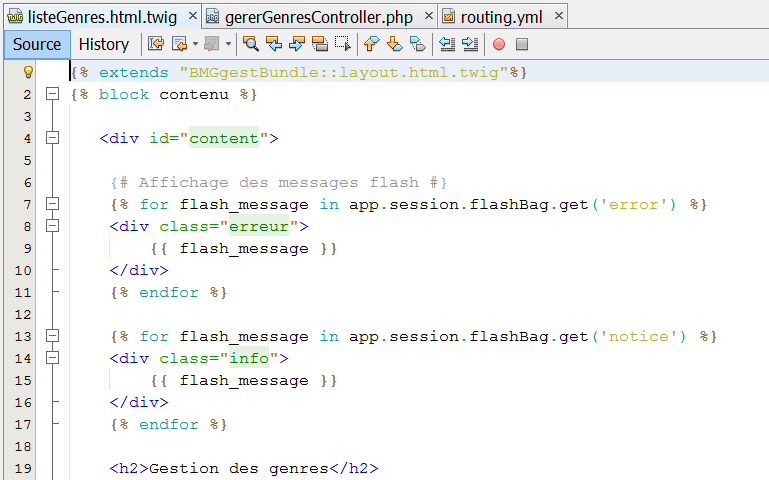
* consulterAction($id)
* ajouterAction()
* supprimerAction($id)
* modifierAction($id)

Pour le moment, nous coderons la fonction consulterAction($id).

Deux cas d'erreur ont été traités :

* L'URL saisie est consulterGenre : il manque l'ID (message = "Ce genre n'existe pas !")
* L'ID passé dans l'URL n'existe pas : message = "Aucun genre n'a été transmis pour consultation !"

Ces deux cas vont produire l'affichage d'un message flash dans la vue listeGenres.html.twig sur laquelle l'utilisateur sera redirigé. Pour afficher ces messages flash, ajouter le code suivant dans la vue listeGenres.html.twig.



On obtient dans chacun des deux cas d'erreur évoqués ci-dessus les résultats suivants :

|  |  |
| --- | --- |
|  |  |

Le code de la fonction consulterAction() est donné ci-dessous :

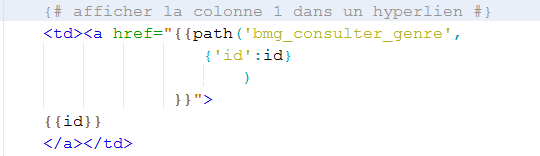


* ligne 181 : n'y a pas d'erreur, on a pu récupérer les informations sur le genre dans l'objet $leGenre grâce au repository et à la méthode find($id).
* ligne 210 : pour l'afficher, on appelle la vue consulterGenre.html.twig.

### La vue consulterGenre.html.twig

**La vue listeGenres.html.twig**

Dans un premier temps, modifions la vue *listeGenres.html.twig* et codons **le lien** permettant de consulter un genre. A la place de la ligne 36 (<td> {{id}} </td>) ajouter le code suivant :



**Remarques :**

* Le lien s'écrira de la même façon que précédemment, avec une nouveauté : l'ajout d'un **paramètre** dans l'URL (id du genre). En cliquant sur l'ID "BIO", l'URL sera :



**La vue *consulterGenre.html.twig***

Le code est donné ci-dessous :

|  |  |
| --- | --- |
|  | On peut remarquer en lignes 22, 23 et 24 le codage des liens pour l'ajout, la modification et la suppression d'un genre, avec un paramètre id pour les deux derniers.  Ligne 40 : l'affichage {{}} du libellé du genre, en faisant appel à **l'accesseur de l'objet Genre, crée par Doctrine.**  Avec **Twig**, la notation se fait avec un point. En php, ce serait :  leGenre->getLibGenre() |

### Les tests

* consulterGenre/XXX
* consulterGenre /ROM
* consulterGenre

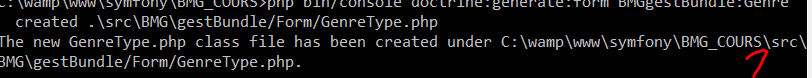
## Ajouter un genre

Pour ajouter un genre, nous allons générer un constructeur de formulaire Symfony, lié à l'entité Genre.

En ligne de commande, taper la commande suivante :

php bin/console doctrine:generate:form BMGgestBundle:Genre

Un fichier *GenreType.php* a été créé dans un nouveau dossier *Form*.



Remplacer le code de la fonction BuildForm() par ce code :

/\*\*

\* {@inheritdoc}

\*/

public function buildForm(FormBuilderInterface $builder, array $options)

{

$builder

->add('codeGenre',TextType::class)

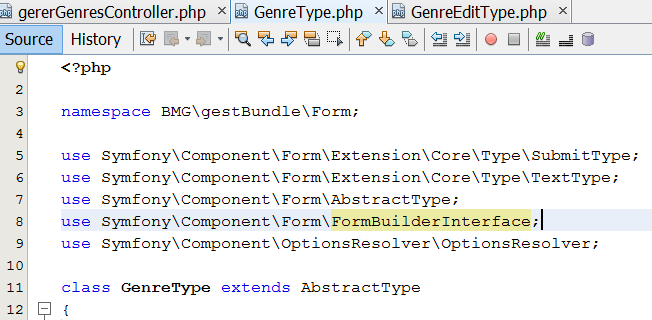
->add('libGenre',TextType::class)

->add('save',SubmitType::class)

;

}

Pour utiliser les classes TextType et SubmitType définissant les contrôles de formulaires, il faudra ajouter les bibliothèques respectives dans *genreType.php*.

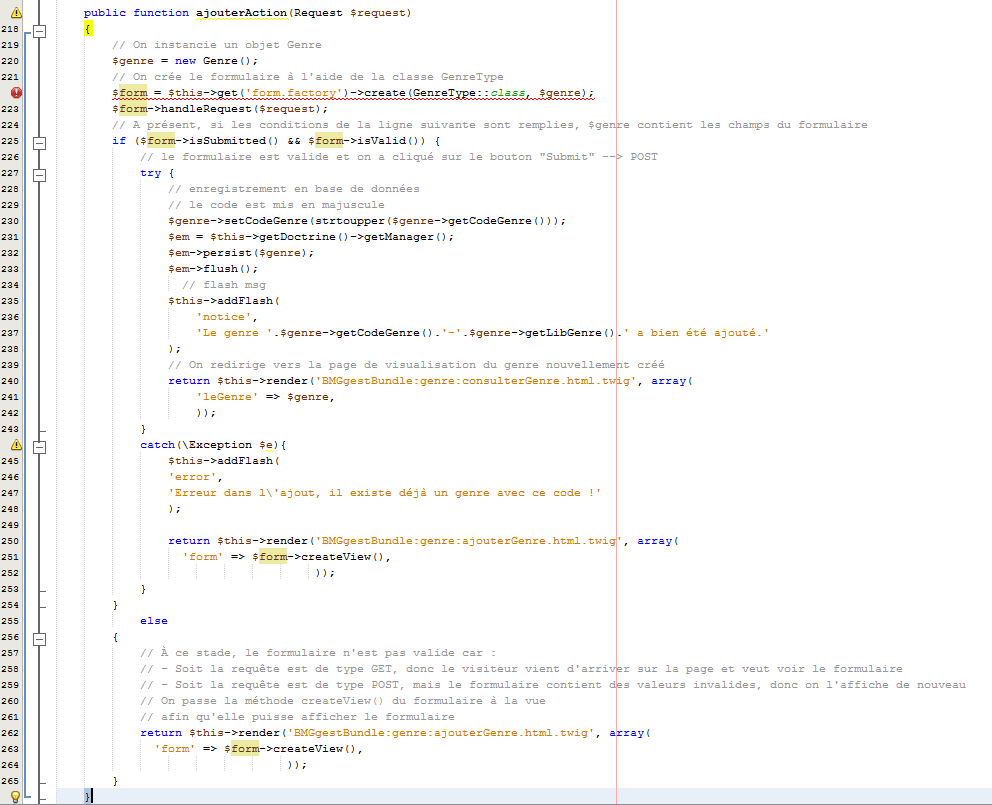


### Le contrôleur : la méthode ajouterAction()

Dans le contrôleur, pour utiliser notre constructeur de formulaire, il faudra ajouter la classe :



Le code de la méthode est donné ci-dessous :



**Remarques**

217 : Observer l'objet Request passé en paramètre : il nous servira ligne 223.

220 : instanciation d'un objet Genre,

222 : création d'un formulaire basé sur GenreType. L'objet Genre instancié en ligne 220 récupérera les valeurs saisies,

223 : si l'on est en POST, l'objet Genre contient les valeurs saisies par l'utilisateur à la fin de cette instruction.

225 : Est-on POST et est-ce-que le formulaire est valide ?

231 à 233 : enregistrement en base de données.

235 à 242 : création d'un message flash de réussite et redirection vers la page de consultation du genre ajouté,

243 à 252 : redirection vers la page *ajouterGenre.html.twig* avec un message flash d'erreur : "Erreur dans l'ajout, il existe déjà un genre avec ce code !".

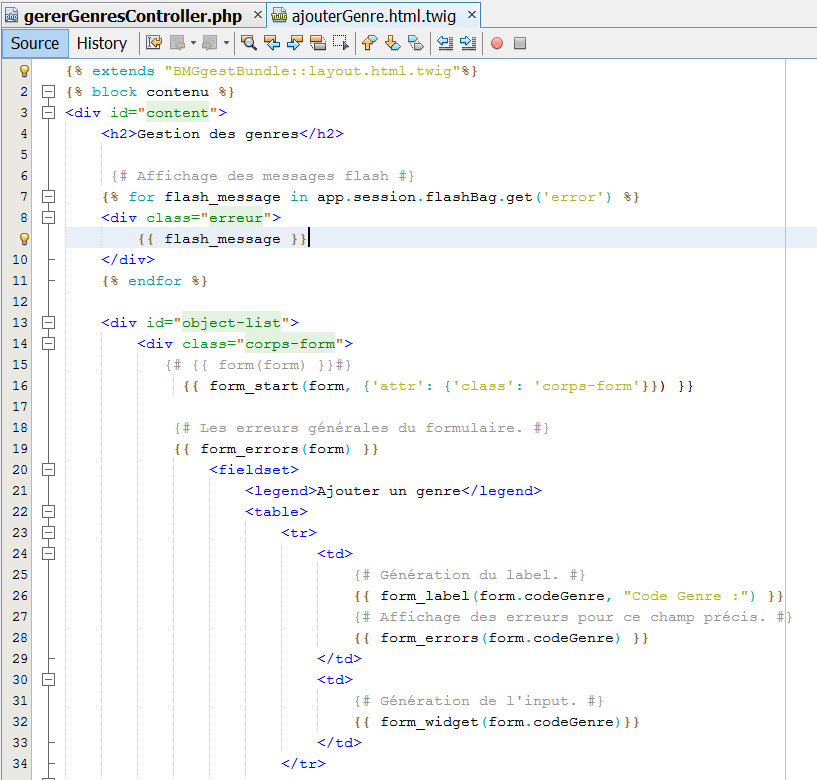
262 : On peut observer le paramètre qui a été passé à la vue : $form->createView()

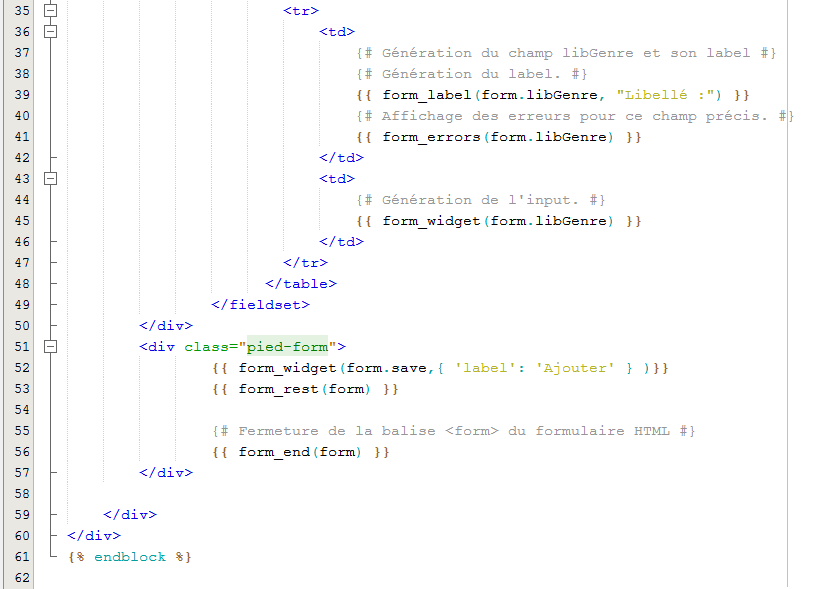
### La vue ajouterGenre.html.twig

La vue doit permettre d'afficher le formulaire envoyé en paramètre par le contrôleur. Ce code ci-dessous serait suffisant pour l'afficher (à tester avant d'ajouter le restant du code en décommentant la ligne 15 et en mettant le reste entre commentaire) :

{{ form(form) }}

Néanmoins, pour respecter la charte graphique, nous détaillerons chaque contrôle à afficher. Le code est le suivant :





### Les tests

* ajouterGenre avec la valeur BIO pour codeGenre
* ajouterGenre avec une valeur inxistante pour codeGenre

## Supprimer un genre

### La méthode supprimerAction() dans le contrôleur

Nous gèrerons 3 messages d'erreur :

* "Suppression impossible : le genre '.$id.' n'existe pas.",
* "Le genre '.$id.' est référencé par des ouvrages, suppression impossible !",
* "Suppression impossible : pas de genre à supprimer !"

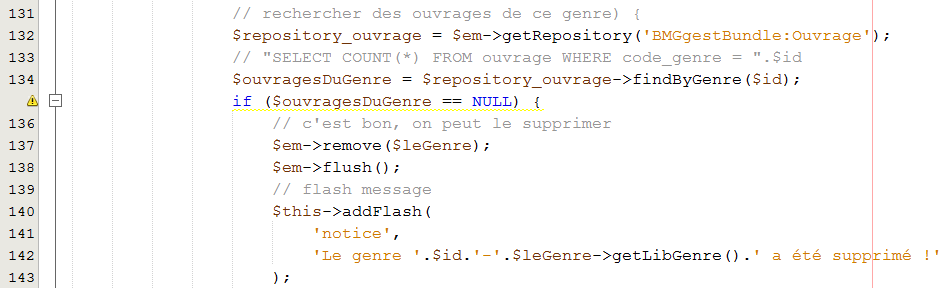
et en cas de réussite le message suivant :

* "Le genre '.$id.'-'.$leGenre->getLibGenre().' a été supprimé !"

Dans tous les cas, une redirection sera effectuée vers la route bmg\_lister\_genres. La vue *listeGenres.html.twig* affichera les messages flash respectifs : info ou erreur.

Le code de la méthode est assez similaire à ce qui a déjà été vu. A vous de l'écrire.

Par contre, la nouveauté concerne la suppression d'un enregistrement dans la base de données via Doctrine. Ce code est donné ci-dessous :



**Remarques :**

ligne 134 : via le repository Ouvrage et la fonction magique "findByNomAttribut", on récupère dans une variable $ouvrageDuGenre tous les ouvrages qui ont pour genre le genre d'id $id ; rappel : $id est le paramètre de la méthode supprimerAction($id).

ligne 137 : si cette variable est nulle, on peut supprimer le genre en utilisant l'entity manager (qui avait été créé plus haut dans la fonction : $em = $this->getDoctrine()->getManager();)

ligne 138 : Il est inutile de persister puisque l'objet $leGenre a été récupéré grâce via Doctrine.

### Les tests

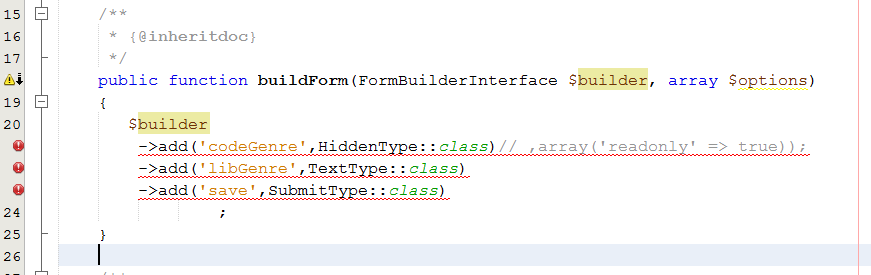
* supprimerGenre/XXX
* supprimerGenre/ROM
* supprimerGenre
* tester la suppression d'un genre existant non référencé par des ouvrages

## Modifier un genre

### Création d'un constructeur de formulaire *GenreEditType.php*

Tout d'abord, pour modifier un genre, nous allons créer un nouveau constructeur de formulaire, différent du formulaire d'ajout : en effet, l'identifiant d'un genre ne doit pas pouvoir être modifié.

Copier le fichier *GenreType.php* et le coller en le renommant *GenreEditType.php.* Vous ne changerez que le code de la fonction buildForm():



N'oubliez pas d'ajouter ce use dans ce fichier :



### La méthode modifierAction() dans le contrôleur

Avant tout, dans le contrôleur ajouter ce use :

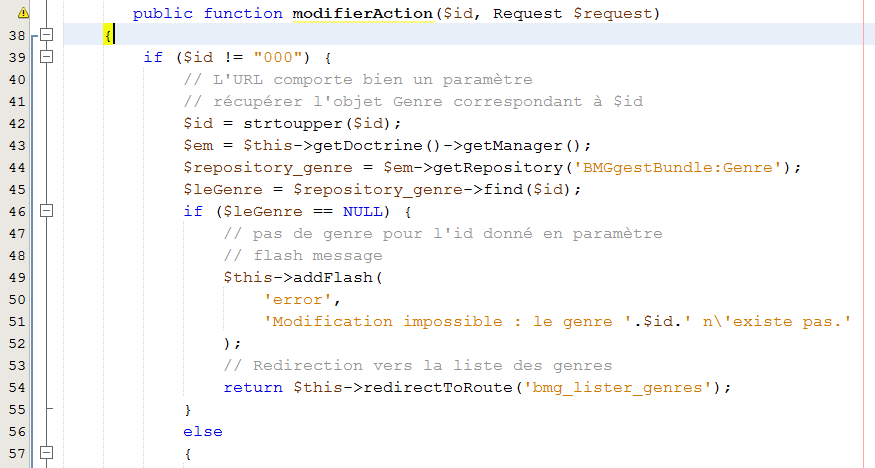
use BMG\gestBundle\Form\GenreEditType;

Nous gèrerons 3 messages d'erreur :

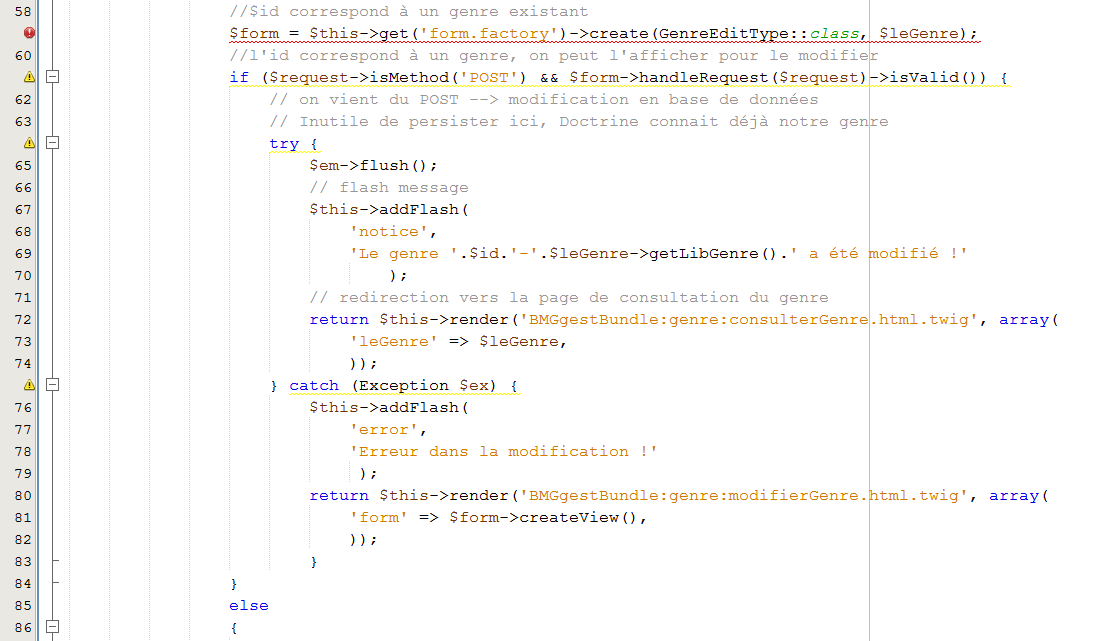
1. "Modification impossible : le genre '.$id.' n'existe pas.", puis redirection vers la liste des genres
2. "Modification impossible : pas de genre à modifier !" puis redirection vers la liste des genres

et en cas de réussite le message suivant :

"Le genre '.$id.'-'.$leGenre->getLibGenre().' a été modifié !" puis redirection vers la consultation du genre modifié.



Pas de difficulté particulière : détection de l'erreur N°1.



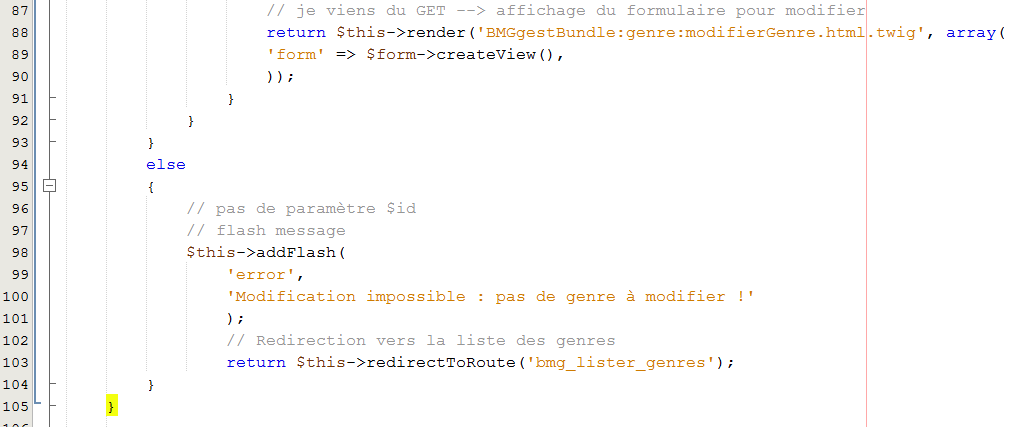
59 : création d'un formulaire basé sur GenreEditType et l'objet Genre créé en ligne 45,

61 : si l'on est en POST et si les valeurs saisies dans le formulaire sont valides, l'objet Genre contient les valeurs saisies par l'utilisateur.

65 : enregistrement en base de données dans un "try".

67 … : puis création d'un message flash de réussite et redirection vers la page de consultation du genre modifié,

76 : si erreur, redirection vers la page *modifierGenre.html.twig* avec un message flash d'erreur : "Erreur dans la modification ".



88 : on vient du GET, on redirige vers la vue qui contient le formulaire de modification.

### La vue *modifierGenre.html.twig*

Cette vue est très similaire à la vue *ajoutGenre.html.twig*.

Il vous reste à l'écrire en vous basant sur cette vue (rappel : on n'affiche pas le code du genre).

### Les tests

* modifierGenre/XXX
* modifierGenre/ROM
* modifierGenre

# Conclusion

Ce tutoriel sur la gestion des genres de l'application BMG est à présent terminé. Vous avez toutes les billes en main pour terminer ce projet Symfony seuls, c'est-à-dire :

* Coder la gestion des auteurs,
* Coder la gestion des ouvrages
* Coder la gestion des utilisateurs avec le FosUserBundle, (avec l'aide d'OpenClassroom …)
* Tester l'application en mode production.

Bon courage !