---------

SYMFONY :

---------

SOMMAIRE :

INTRO : Qu'est-ce que Symfony

ETAPE 1 : Installation de SF 3.4

ETAPE 2 : Les bundles

ETAPE 3 : Les routes et les controllers

ETAPE 4 : Créer la Boutique sur SF

ETAPE 5 : TWIG (moteur de template)

ETAPE 6 : Les assets

ETAPE 7 : Entités

ETAPE 8 : DOCTRINE

ETAPE 9 : Les formulaires

ETAPE 10 : Validation des données

ETAPE 11 : Association Mapping

ETAPE 12 : Sécurité et Utilisateurs

ETAPE 13 : Installation et boutique sur SF 4.3

BONUS : Mise en prod

BONUS : Formulaire de contact

--------------------------------------

INTRO : Qu'est-ce que Symfony

---------------------------------------

1/ Quel intérêt d'utiliser un Framework

A. Une organisation optimisée

B. Fonctionnalités communes à tous les projets

C. Services disponibles (Routing, Sécurité, BDD, Moteur de template, formulaire...)

D/ Communauté

2/ Choix du Framework

A. Propre Framework (pédagogie)

B. Les frameworks fullstack (Symfony, Zend, Laravel, cake...)

C. Les minis Frameworks (Silex, Slim, CodeIgniter, Lumen...)

3/ Symfony

-> Framework Français créé par sensiolabs

-> versions :

LTS (Long Time Support):

v2.8

v3.4 : 2.8 plus souple (moins rigide), avec des fonctionnalités

v4.4 : 3.4 BundleLess, Flex, Encore (webpack), Maker, PHP7.1

----------------------------------

ETAPE 1 : Installation de SF 3.4

Sommaire :

1/ Installer Composer

2/ Installer SF3.4

3/ Arborescence des dossiers et fichiers

4/ Lancement de l'application

------------------------------------

1/ Installer Composer

Composer est un outils de dépendance. Il permet de télécharger et de mettre à jour tous les outils (services, composants, dépendances) dont on a besoin.

- Télécharger : <https://getcomposer.org/download/>

composer-Setup.exe

- Installer : Suivre les étapes.

----------------------------------------------

ETAPE 4 : Créer la Boutique sur SF 3.4

Sommaire :

1/ Création du projet

2/ Réorganiser le AppBundle

3/ Création des première route

1/ Création du projet

- Dans le dossier Symfony

<cmd>

composer create-project symfony/framework-standard-edition Boutiques3

cd Boutique3

php bin/console server:run

Test : localhost:8000

2/ Installer SF3.4

- Dans le dossier Symfony

- MAJ + clic-droit : Ouvrir une fenêtre powershell ici

<cmd>

composer create-project symfony/framework-standard-edition test

==> A ce stade un dossier test a été créé. C'est un nouveau projet SF.

3/ Arborescence des dossiers et fichiers

- app/ : contient toute la configuration de l'app (BDD, sécurité, routes...)

- bin/ : Les éxécutables de l'app (de SF)

- src/ : Le dossier dans lequel nous allons coder notre MVC

- var/ : Les fichiers écrits par SF au fur et à mesure (cache, logs)

- tests/ : Les tests unitaires.

- web/ : repertoire WEB (app.php ou app\_dev.php, img/css/js/fonts)

- vendor/ : Le coeur de SF (les fichiers codés par les sensio Labs)

- composer.json : Contient la liste de toutes les dépendances dont on a besoin.

4/ Lancement de l'application

- Méthode 1 :

localhost/Symfony/test/web/app.php

localhost/Symfony/test/web/app\_dev.php

- Méthode 2 :

On va dans le dossier test/

<cmd>

php bin/console server:run

localhost:8000

- Différence entre mode prod et dev :

localhost/Symfony/test/web/toto

localhost:8000/toto

===> A ce stade notre application affiche la page de bienvenue. Cela signifie que tout va bien.

-----------------------

ETAPE 2 : Les bundles

Sommaire :

1/ Le concept des Bundles

2/ Création de notre premier Bundle

-----------------------

1/ Le concept des Bundles

- Les bundles sont des briques de notre applications.

UserBundle :

UserController (C)

UserModel (M)

View : inscription/connexion/profil...

Avec la nouvelle version de SF (4), on considère qu'il est préférable de créé un seul Bundle (AppBundle).

2/ Création de notre premier Bundle

On va créer un Bundle : POLES/TestBundle

<cmd>

php bin/console generate:bundle

-> Yes

-> POLES/TestBundle

-> POLESTestBundle

-> src/

-> annotation

==> Notre Bundle a été créé, mais il faut l'enregistrer.

<code>composer.json

"psr-4": {

"AppBundle\\": "src/AppBundle",

"POLES\\": "src/POLES"

},

- Mise à jour de l'app

<cmd>

composer update

===> Notre Bundle existe dans src/POLES/TestBundle

Il est composé de 4 dossiers :

- Controller : les controllers du bundle

- DependencyInjection : Injection de dépendances.

- Resources : Vues et Routes (YML) et enventuellement JS

- Test : Test liés aux fonctionnalité du Bundle.

===> A ce stade le Bundle est fonctionnel, mais il y a une petite erreur qu'on va corriger.

- test/src/POLES/TestBundle/Controller/DefaultController.php

<code>

return $this->render('POLESTestBundle:Default:index.html.twig');

Devient :

return $this->render('@POLESTest/Default/index.html.twig');

===> A ce stade la page d'accueil affiche "Hello World".

--------------------------

ETAPE 3 : Les routes et les controllers

Sommaire :

1/ création de routes

2/ L'objet Request

3/ L'objet Response

4/ Redirection

5/ Message

--------------------------

1/ création de routes

("/") -> route simple homepage

("/bonjour/") -> route echo (erreur)

("/bonjour2/") -> route response

("/hello/{prenom}") -> route response + param URL

("/hola/{prenom}") -> route render de vue (html.twig) + param URL

("/ciao/{prenom}/{age}") -> route render de vue + 2 params URL

("/redirect/") -> route avec redirection (RedirectResponse)

("/redirect2/") -> route avec redirection (redirectToRoute())

("/message/") -> route avec redirect et message en session

2/ L'objet Request

<code>

use Symfony\Component\HttpFoundation\Request;

Correspond à la partie requete de la requête HTTP.

<code>

$session = $request -> getSession();

-> récupère la session ($\_SESSION)

Autres exemples :

<code>

$session -> set('prenom', 'Yakine');

echo $session -> get('prenom');

Autre méthode :

<code>

$request -> session -> set('prenom', 'Yakine');

Autres utilités de $request :

<code>

$request -> query -> get('argument\_url');

$request -> request -> get('champs\_form');

$request -> cookies -> get('cookie');

$request -> server -> get('server');

3/ L'objet Response

- Correspond à la partie réponse d'une requête HTTP.

<code>

use Symfony\Component\HttpFoundation\Response;

- Toute action (fonction) doit avec une réponse.

<code>

return new Response('toto');

- Même la fonction render est une réponse.

<code>

return $this -> render();

return $this -> getTemplating() -> renderResponse();

4/ Redirection

<code>premiere méthode :

use Symfony\Component\HttpFoundation\RedirectResponse;

cf route "/redirect/" et "/redirect2"

/!\ A ce stade toutes nos routes doivent avoir un name

5/ Message

<code>

$session -> getFlashBag() -> add();

-> permet d'enregistrer des messages qui seront disponibles dans d'autres pages (messages de félicitations, ou d'erreur).

-> app.session : Qui nous permet en Twig de récupérer les infos en session.

-> app.user : Qui nous permet en Twig de récupérer les infos de l'utilisateur actuellement connecté

---------------------------------------------------

ETAPE 4 : Créer la boutique sur sf 3.4

1/ Création du projet

2/ Réorganiser le AppBundle

3/ Création des premieres route

--------------------------------------------

--------------------------------------------

1 / Création du projet

- Dans le dossier Symfony

<cmd>

composer create-project symfony/framework-standard-edition Boutiques3

cd Boutique3

php bin/console server:run

Test : localhost:8000

2 / Réorganiser le AppBundle

localhost/ -> index.html.twig

localhost/categorie/pull-> index.html.twig

localhost/produit/12 ->show.html.twig

1/ Réorganiser le AppBundle

-> Création de nos controleurs:

-AdminController

-MembreController

-CommandeController

-ProduitController

-> Création de nos vues:

-Admin/ (list\_produit, form\_produit etc...)

-Membre/ (connexion, deconnexion, inscription, profil...)

-Commande/ (panier, transport, livraison, paiement...)

-Produit/ (home, categorie, boutique, fiche\_produit...)

3/ Création des premières routes

-> ProduitController : "/", "/produit/{id}/" etc...

->AdminController : "/admin/produit/", "/admin/membre/" etc...

-------------------------------------

ETAPE 5 : TWIG(moteur de template)

Sommaire

1/ Création du layout

2/ L'héritage twig

3/ Modification de nos vues

4/ Documentation Twig

---------------------------------------

TWIG est le moteur de template de Symfony. Un moteur de template (tpl, smarty, twig, blade.....) permet d'afficher du php dans les fichiers HTML de manière plus simple

exemple:

<?= $membre['prenom']?>

{{ membre.prenom }}

1/ Création du layout

Chemin : Boutique3/app/Ressources/views/layout.html.twig

Un layout est une structure de page, dans laquelle on peut rajouter des elements/du contenu.

le concept des layout permet d'avoir plusieurs structures de pages dans notre site (ex: front vs backoffice).

2/ L'héritage twig

Avec Twig on parle d'heritage. Cela signifie que nos vues héritent (extends) d'un layout, et peuvent contenir de l'HTML à l'interieur des blocks disponibles (créés dans le layout).

======> e manière générale, il faut voir le fonctionnement de Twig et de la méthode render comme un chargement du contenu "empilée" (analogie de la poupée russe). Render temporise (mémoire tompon) le contenu, affiche au meme temps le layout , la vue et les paramètres.

3/ Modification de nos vues

<code>

{% extende 'layout.html.twig' %}

{% block content %}

contenu HTML de la page

{% endblock %}

/!\ ATTENTION : quand on utilise l'heritage Twig, nos vues ne peuvent contenir de l'HTML qu'à l'interieur des blocks prévus pour.

4/ Documentation Twig

<https://twig.symfony.com/doc/2.x/>

======> A ce stade notre app permet d'afficher les pages accueil, connexion, inscription, profil, boutique, produit, categorie, gestion membre; gestion produit, gestion commande

======> Par contre nous ne sommes pas en mesured'echanger avec la BDD (ajouter un contenu, recuperer un contenu), il faut voir les entity et Doctrine

-------------------------------------

ETAPE 6 : Les assets

Sommaire

1/ Modification du fichier composer.json

2/ Mise à jour de l'app

3/ Dossier Web/ (photo, css/js....)

4/ Modification des vues

---------------------------------------

- Le composant Asset de SF nous permet de gérer les ressources (photo, img, js, fonts, css...) et de les gerer de manière absolue.

1/ Modification du fichier composer.json

<code> composer.json

"require":{

"symfony/asset": "^3.4"

}

2/ Mise à jour de l'app

<cmd>

compser update

3/ Dossier Web/ (photo, css/js....)

Boutique3/web/

photo/

css/

js/

4/ Modification des vues.

href="../css/style.css"

-> href="{{ asset('css/styles.css') }}"

href="inscription.php"

href="{{ path('inscription') }}"

la fonction asset() de TWIG nous permet de charger une ressource (photo, css, js, fonts)

la fonction path() de TWIG nous permet de créer un lien vers une page (route).

// Liens dynamiques :

src="../../photo/<?= $pdt['photo']?>"

src="{{ asset('photo/' ~ pdt.photo) }}"

href="fiche\_produit.php?id=<?=$pdt['id'] ?>"

href="{{ path('produit', {'id': pdt.id }) }}"

@Route("/produit/{id}", name="produit")

explication:

produit======>nom de la route

id===========>paramètre dynamique de la route

pdt.id=======>valeur du paramètre dynamique

---------------------------------

href="boutique.php?categorie=<?=$pdt['categorie'] ?>"

href="{{ path('categorie', {'cat': cat.categorie }) }}"

@Route("/categorie/{cat}", name="categorie")

-------------------------------------

ETAPE 7 : Les Entitées

Sommaire

1/ Doctrine ORM et le code des Entités (Entity)

2/ Créer Entités Produit

3/ Annotation

4/ Mattre à jour

5/ Générer les Entités en ligne de Commande

6/ Générer la BDD via les Entités

Générer les entités depuis la BDD

----------------------------------

1/ Doctrine ORM et le code des Entités (Entity)

- D'une certaine manière, les entités correspondent à la partie MODEL de notre MVC. c'est la relation avec la BDD, mais en PHP et non en SQL.

- Théoriquement, nous n'avons plus besoin du SQL

- ORM: Object Relation Mapping nous allons creer nos entity sous forme de classes PHP ( Plain old PHP Objest), qui vont permettre à doctrine de conprendre notre base de données et de la manipuler.

Par exemple: pour faire INSERT INTO Produit

$produit = new Produit;

persist($produit);

remove($produit);

createForm($produit);

===> Pour que cela soit possible, il faut expliquer à Doctrine quel est la relation entre la BDD et nos entités

2/ Créer Entités Produit

A. Créer le dossier src/AppBundle/Entity/

B. Créer le fichier produit.php

C. Créer notre class Produit et nos proprietés et les getteurs et les setteurs

3/ Annotation (mapping)

- Avec Doctrine ORM, on paramètre le mapping via les annotations.

use Doctrine\ORM\Mapping as ORM

--> Voir le Fichier Entity/Produit.php

/!\ ATTENTION: les annotations pour L'ID (primary key ) sont plus longues.

Liens :

Basic mapping

<https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/basic-mapping.html>

Association Mapping

4/ Mattre à jour

<cmd>

php bin/console doctrine:schema:update --dump-sql

--> voir la requete

php bin/console doctrine:schema:update --force

--> executer les modifs en BDD

==> Permet aussi de créer une BDD à partir de nos entités

5/ Générer les Entités en ligne de Commande

A. on doit connecter notre App à la BDD

- app//config/parameters.yml

- php bin/console doctrine:generate:entity

AppBundle:Membre

-> on suit les etapes.

====> cela a créé le fichier Entity/Membre.php en quelques lignes de commandes.

6/ Générer la BDD via les Entités

-on peut crée la BDD, depuis les entités déja codées

<cmd>

php bin/condole Doctrine:schema:update --force

---> créer les tables dans la BDD . par contre il faut que la BDD existe déja

7/ Générer les entités depuis la BDD

-on peut crée la BDD, depuis les entités depuis la BDD

<cmd>

php bin/condole doctrine:mapping:import AppBundle\Entity annotation --path=src/AppBundle/Entity

===> Cette commande nous a permis de generer les Entités depuis les tables de la BDD .

- Par contre celan'a ni généré les repository, ni les getteurs ni les setteurs .

<code>

@ORM\Entity(repositoryClass="AppBundle\Repository\ProduitRepository")

@ORM\Entity(repositoryClass="AppBundle\Repository\CommandeRepository")

@ORM\Entity(repositoryClass="AppBundle\Repository\MembreRepository")

@ORM\Entity(repositoryClass="AppBundle\Repository\DetailsCommandeRepository")

<cmd>

php bin/condole doctrine:generate:entities AppBundle

==> cela nous crée nos getter et nos setter et génére les repository

-------------------------------------

ETAPE 8 : Doctrine

Sommaire

1/ LE SERVICE dOCTRINE

2/ Acceder au service doctrine depuis les controllers

3/ Requets select \* FROM....

4/ Requets select \* FROM....WHERE id =

5/ Requets select \* FROM....WHRE .... =......

6/ Requete INSERT / UPDATE

7/ REQUETE DELETE

8/ Create query et query builder

---------------------------------------

1/ Le service doctrine

Doctrine fait 2 choses

1/ Doctrine ORM (Objet Relation Mapping) :

il permet de lier les tables de notre BDD à nos entitées, grace à ca on ne va plus manipuler la BDD mais des objets (la table Produit ---> Entity Produit)

Doctrine DBAL (DataBase Abstract Layer)

le DBAL c'est une couche qui passe au-dessus de PDO. A partir de maintenant on ne fait plus de requête SQL, mais on va manipuler du PHP.

- En résumé le DBAL va nous permettre de faire des SELECT, INSERT, UPDATE, DELETE, non pas via du SQL mais via des fonctions PHP . On parle de DQL (Doctrine Query Language)

----> NOUS NE FERONS PLUS DE SQL (OU PRESQUE)

2/ Acceder au service doctrine depuis les controllers

A. il faut pouvoir manipuler les entités dont on a besoin

<code>

use AppBundle\Entity\Produit;

B. Acceder au repository dans un controller :

<code>

$repo = $this-> getDoctrine -> getRepository(Produit::class);

C. Acceder au manager (qui peut faire des requetes sur toutes les tables)

$em = $this ->getDoctrine -> getManager();

3/ Requets select \* FROM....

Depuis le repository

<code>

$repo = $this-> getDoctrine -> getRepository(Produit::class);

$produits = $repo -< findAll();

4/ Requets select \* FROM....WHERE id =

Depuis le repository

<code>

$repo = $this-> getDoctrine -> getRepository(Produit::class);

$produits = $repo -> find(id);

depuis le manager:

<code>

$em = $this ->getDoctrine -> getManager();

$produits = $repo -> find(Produit::class, $id);

5/ Requetes SELECT \* FROM ... WHERE ... = ...

La fonction findBy () va nous permettre de récupérer des données de manière plis ciblée.

<code>

$repo = $^this ->getDoctrine() ->getRepositoryProduit::Class;

$produit = $repo ->findBy(array('categorie') => $cat));

$produit = $repo -> findBy(array('taille' => 'l'));

$produit = $repo -> findBy(array(

'categorie' => 'tee-shirt',

'taille' => 'l'

));

SELECT $ FROM produit WHERE taille = 'l' AND categorie = 'tee-shirt'

$produit = $repo -> findBy(['couleur' => 'rouge'], ['prix' => 'DESC'], 0, 10)

SELECT \* FROM produit WHERE couleur = 'rouge' ORDER BY prix DESC LIMIT 0, 10

$produit = $repo -> findOneBy(array('taille' => 'l'));

SELECT \* FROM produit WHERE taille = 'l', LIMIT 0, 1

6/ Requete INSERT / UPDATE

avec doctrine on manipule des entités des objet

-> enregistrement:

- on cree un objet vide

- on hydrate l'objet

- on persiste et on flush

<code>

$produit = new Produit;

$produit -> setTitre('hjhjghj')

$produit -> setReference('ghhgd')

...

$em -> persist($produit) ;

$em -> flush();

-> Modification:

pour update une entréé c'est la meme chose que pour ADD, à ceci près que l'objet n'est pas vide à la base

7/ REQUETE DELETE

- Avec doctrine on manipule des objets (Entités) donc on va récuperer d'abord l'objet à supprimer avec la fonction find().

<code>

$em->remove();

$em->flush();

SELECT DISTINCT(categorie) FROM produit

8/ Create query et query builder

Pour effectuer des requêtes spécifiques, nous avons 2 possibilités :

- Create Query (SQL)

- Query Builder (PHP)

==> VOIR Produit Repository

/!\ Dans l'objectif de "factoriser" notre code nous créons nos requetes spécifiques dans le repository, afin de les utiliser de manière simple dans les controller

==> Voir le ProduitController, route "/categorie/" et la fonction GetAllCategories() déclarées dans le ProduitRepository.

\*\*\*\*\*\*\*\*\*\*\*\*\*

A faire

\*\*\*\*\*\*\*\*\*\*\*\*\*

la gestion des produits

AdminController

Affichage des produits :

-> /admin/produit/

$produits qui va recuperer tous les produits du site

-> list\_produit.html.twig

-> tableau avec tous les infos des produits

-> action : modifier / supprimer

-------------------------------------

ETAPE 9 : Les formulaire

Sommaire

1/ Le fonctionnement des formulaires

2/ Généréer des formulaires

3/ Recuperer les donnees du formulaire

4/ Persollaliser le formulaire avec bootstrap

5/ update un enregistrement

6/ validation des données

7/ Champs file

---------------------------------------

1/ Le fonctionnement des formulaires

De la meme manière qu'on ne manipule pas des enregistrements en BDD, mais bies des objets (Entity), chaque foemulaire va être lié à une entité

- Ainsi un formulaire est lié à un objet..

Inscription---------------> objet Membre

Ajout de produit----------> objet Produit

Modif de produit----------> objet Produit

Un formulaire est généré grâce à une classe (Type--> ProduitType) chaque champs d'un formulaire correspond à une classe, service de SF. Cela permet de générer tout les controles sur les foemulaires.

<cmd>

php bin/console generate:doctrine:form AppBundle:Produit

===> cela créé un dossier form dans notre bundlle, et un fichier ProduitType.php, qui contient la construction du formulaire.

3/ Recuperer les donnees du formulaire

A. Générer le form dans la route

<code>

use AppBundle\Form\ProduitType

$form = $this -> createForm(ProduitType::class, $produit);

B. Afficher le formulaire dans la vue

- on envoie le formuliare à la vue via les params

<code>

$params = array('produitForm' => $form -> CreateView())

- On affiche le formulaire :

<code>

{{form(produitForm) }} -->affiche tout le formulaire

---

{{ form\_row(produitForm.reference) }} --> affiche un champs (label, champs, errors)

----

{{ form\_label(produitForm.reference) }} --> affiche le label

{{ form\_widget(produitForm.reference) }} --> affiche le champs

{{ form\_errors(produitForm.reference) }} --> affiche les erreurs

/!\ Si on affiche les champs un par un , il ne faut pas oublier :

{{ form\_start (produitForm )}}

{{ form\_end (produitForm )}}-

C. Recuperer les données saisies

<code>

$form -< handleRequest($request);

/!\ cette ligne est importante pour pouvoir récuperer les données en POST !!

D. Checker la validite du form et traiter les données (enregistrement)

<code>

if($form -> isSubmitted() && $form -> is valid())

{

//persist

//flush

//message

//redirection

}

4/ Personaliser le formulaire avec bootstrap

<code> app/Congif/config.yml

twig:

form\_themes:

- 'bootstrap\_4\_layout.html.twig'

5/ Update un enregistrement

pour update un produit avec e système des formulaires sur SF, rien de plus simple. Lorsqu'on crée le formulaire dans la route, on precise un objet déja existant (cf route admin/produit/update/{id}).

-----> on dit qu'on hydrate le formulaire

Exercice

A faire: Inscription (page inscription)

1/ Entity Membre

2/ Controller : MembreController ----> Route "/inscription"

3/ Formulaire (MembreType) --> console

4/ Route : Créer le formulaire et sa vue

5/ La vue : Afficher le formulaire

6/ Route : Récupérer les infos du formulaire et enregistrer le membre .

Attention : Etant donnée que le champs Statut existe dans notre BDD, le formulaire va être créer avec un champs Statut, hors ce n'est pas au membre de choisir son statut

6/ validation des données

- De base le formulaire est généré par SF de manière basique. Pour gérér nous memes les contraintes, et la securité cela se passe dans le fichier ProduitType.php, et on utilise constraints pour definir les contraintes liées au formulaire

-> cf ProduitType.php

7/ Champs file

Pour la gestion des photos il faut:

1/. Enregistrer le nom de la photo dans la BDD

2/. Enregistrer la photo (data) sur le server.

==> CF fonction uploadFile dans l'entity Produit.

/!\ dans notre formulaire d'ajout de produits , il n'est plus necessaire d'avoir un champs photo (TEXT), mais un champs file (FILE) qui va recuperer les infos de la photo uploadée.

Pour aller plus loin avec les formulaires st symfony

<https://symfony.com/doc/current/reference/forms/types.html>

-------------------------------------

ETAPE 10 : Validation des données (Assert)

---------------------------------------

ETAPE 11 : Association mapping

-------------------------------------

ETAPE 12 : Sécurité et Utilisateurs

Sommaire

1/ Niveau de securite

2/ Fichier de securite

3/ Creation des routes

4/ Classe Membre (extends UserInterface)

5/ Layout

---------------------------------------

1/ Niveau de securite

De base SF va considerer un visiteur lambda comme un anonyme.

- ensuite on va définir des niveaus de securité de l'utilisateur connecté:

- ROLE\_USER (user connecté)

- ROLE\_ADMIN

- ROLE\_SUPER\_ ADMIN

- SF a un système de parefeu :

- On definit les routes que les ROLE peuvent emprunter.

-> si un user anonyme --> hemepage = ok

-> si un user anonyme --> profil = pas ok -> connexion -> profil

-> si un user connecté --> profil = ok -> connexion -> profil

-> si un user connecté --> admin/produit = pas ok -> page erreur(404)

En fonction du visiteur et de la ressource demandée on peut paramétrer une réaction

2/ Fichier de securite

3/ Creation des routes

4/ Classe Membre (extends UserInterface)

5/ Layout

-------------------------------------

ETAPE 12 :

Sommaire

1/

2/

3/

4/

---------------------------------------