1. QU'EST CE QUE SYMFONY

1.1. Symfony est un framework

Le terme framework est fréquemment utilisé dans des contextes différents mais il peut être traduit par **cadre de développement**.
Les frameworks se présentent sous diverses formes, qui peuvent inclure tout ou partie des éléments suivants :

- un ensemble de classes généralement regroupées sous la forme de bibliothèques pour proposer des services plus ou moins sophistiqués
- un cadre de conception reposant sur les design patterns pour proposer tout ou partie d'un squelette d'applications
- des recommandations sur la mise en oeuvre et des exemples d'utilisation
- des normes de développement
- des outils facilitant la mise en oeuvre

L'objectif d'un framework est de faciliter la mise en oeuvre des fonctionnalités d'un domaine d'activité. Il doit permette au développeur de se concentrer sur les tâches spécifiques à l'application à développer plutôt qu'à des tâches techniques récurrentes telles que :

- l'architecture de base de l'application
- l'accès aux données
- l'internationalisation
- la journalisation des événements (logging)
- la sécurité (authentification et gestion des rôles)
- le paramétrage de l'application
- ...

La mise en oeuvre d'un framework permet notamment :

- de capitaliser le savoir-faire sans "réinventer la roue"
- d'accroître la productivité des développeurs une fois le framework pris en main
- d'homogénéiser les développements des applications en assurant la réutilisation de composants fiables
- donc de faciliter la maintenance notamment évolutive des applications

Cependant, cette mise en oeuvre peut se heurter à certaines difficultés :

- le temps de prise en main du framework par les développeurs peut être plus ou long en fonction de différents facteurs (complexité du framework, richesse de sa documentation, expérience des développeurs, ...)
  Le plus gros défaut des frameworks est lié à leur complexité : il faut un certain temps d'apprentissage pour avoir un minimum de maitrise et d'efficacité dans leur utilisation.
- les évolutions du framework qu'il faut répercuter dans les applications existantes

L'intérêt majeur des frameworks est de proposer une structure identique pour toutes les applications qui l'utilisent et de fournir des mécanismes plus ou moins sophistiqués pour assurer des tâches communes à toutes les applications.
Pour choisir un framework, les caractéristiques suivantes doivent être prises en compte :

- adoption par la communauté
- la qualité de la documentation
- le support (commercial ou communautaire)
- le support par les outils de développement

Pourtant le développement d'applications web est relativement compliqué à concevoir et à implémenter pour plusieurs raisons :

- la nature du protocole http (basé sur un modèle simple de request/response sans état)
- les nombreuses technologies à mettre en oeuvre (HTML, XHTML, CSS, JavaScript, ...) ainsi que leurs différentes versions
- le support de ces technologies par les différents navigateurs est particulièrement différent
- HTML est un langage pauvre qui ne permet que le développement de formulaires assez rudimentaire

Les frameworks de développement web permettent généralement une séparation logique d'une application selon le concept proposé par le **modèle MVC (Modèle/Vue/Contrôleur).**

Voici quelques frameworks connus :

- frameworks Java : Struts, Spring, JSF
- frameworks Php : CodeIgniter, cakePHP et Symfony

1.2. Symfony repose sur le modèle MVC

Voir cours précédent sur le diagramme de classes.
Son principal intérêt est la **séparation des données** (*modèle*), de l'**affichage** (*vue*) et des **actions** (*contrôleur*).
L'approche MVC apporte de réels avantages:

- Une conception **claire** et **efficace** grâce à la séparation des données de la vue et du contrôleur. La séparation permet ainsi aux développeurs de faire des modifications sur une partie de l'application sans affecter les autres. Le cas le plus courant est par exemple, que l'on peut réécrire seulement la partie visible aux utilisateurs (interfaces web, mobiles, ..) sans modifier le reste de l'application.
- Un **gain de temps** de maintenance et d'évolution du site (voir ci-dessous)
- Une plus **grande souplesse** pour organiser le développement du site entre différents développeurs (indépendance des données, de l'affichage (webdesign) et des actions)

1.3. Symfony intègre l'ORM Doctrine

La quasi-totalité des applications de gestion traitent des données dans des volumes plus ou moins importants. Dès que ce volume devient assez important, les données sont stockées dans une base de données.

La correspondance des données entre le modèle relationnel et le modèle objet doit faire face à plusieurs problèmes :

- le modèle objet propose plus de fonctionnalités : héritage, polymorphisme, ...
- les relations entre les entités des deux modèles sont différentes
- les objets ne possèdent pas d'identifiant en standard (hormis son adresse mémoire qui varie d'une exécution à l'autre).

- Dans le modèle relationnel, chaque occurrence devrait posséder un identifiant unique

Le mapping Objet/Relationnel (ORM) consiste à réaliser la correspondance entre le modèle de données relationnel et le modèle objets de la façon la plus facile possible.
Un outil d'ORM doit cependant proposer un certain nombre de fonctionnalités parmi lesquelles :

- Assurer le mapping des tables avec les classes, des champs avec les attributs, des relations et des cardinalités
- Proposer une interface qui permette de facilement mettre en oeuvre des actions de type CRUD
- Eventuellement permettre l'héritage des mappings
- Proposer un langage de requêtes indépendant de la base de données cible et assurer une traduction en SQL natif selon la base utilisée
- Supporter différentes formes d'identifiants générés automatiquement par les bases de données (identity, sequence, ...)
- Proposer un support des transactions
- Assurer une gestion des accès concurrents (verrou, dead lock, ...)
- Fournir des fonctionnalités pour améliorer les performances (cache, lazy loading, ...)

Les solutions de mapping sont donc riches en fonctionnalités ce qui peut rendre leur mise en oeuvre plus ou moins complexe.

De nombreuses difficultés peuvent survenir lors de la mise en oeuvre d'un outil d'ORM

- Difficultés à mapper le modèle relationnel à cause de la complexité du modèle ou de sa mauvaise conception
- Temps d'apprentissage de l'outil plus ou moins important
- Difficultés pour mettre en oeuvre les transactions
- Parfois des problèmes de performance peuvent nécessiter un paramétrage plus fin notamment en se qui concerne la mise en oeuvre de caches ou du chargement tardif (lazy loading : chargement en mémoire de ce qui est nécessaire à l'instant t)
- Difficultés pour maintenir les fichiers de configuration souvent au format XML et à les synchroniser avec les évolutions du modèle de données. Des outils existent pour certaines solutions afin de faciliter cette tâche

Les orm les plus connus :

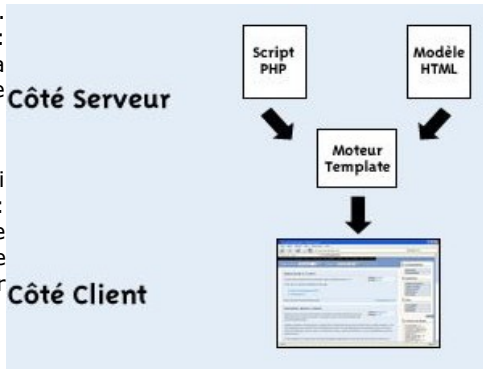
- **ORM Pour Java** : Hibernate est le framework open source de mapping O/R le plus populaire. Cette popularité est liée à la richesse des fonctionnalités proposées et à ses performances.

Hibernate propose son propre langage d'interrogation HQL et a largement inspiré les concepteurs de l'API JPA. Hibernate est un projet open source de mapping O/R qui fait référence en la matière car il possède plusieurs avantages : manipulation de données d'une base de données relationnelles à partir d'objets Java, facile à mettre en oeuvre, efficace et fiable, open source

- **ORM Pour PHP :** Doctrine possédant les mêmes caractéristiques qu'Hibernate mais pour le langage PHP.

1.4. Symfony intègre le moteur de template TWIG

Le terme *Template* peut se traduire par modèle, gabarit ou patron. Les fichiers templates sont des fichiers qui comportent la mise en page des pages. Ces fichiers peuvent portés, a priori, n'importe quelles extensions (html, tpl, txt, php, ..).
L'intérêt majeur d'utiliser un moteur de templates tel que Twig est **de séparer le code PHP de la mise en page HTML.**
Le code PHP dans un fichier, la mise en page contenant les balises HTML dans un autre.
On se retrouve alors avec, d'un coté, le script qui fait tout ce qu'il a à faire (ex: récupération de données dans une base de données, traitement…), et d'un autre coté, la mise en page avec des zones prédéfinies où seront placées les données générées par le script.



L'avantage est de pouvoir travailler uniquement sur la mise en page, sans modifier quoi que ce soit dans le code PHP et inversement, ou de diviser efficacement le travail à faire : le programmeur s'occupant uniquement de la partie scripting, et le designeur/infographiste, de la mise en page. Le schéma ci-contre illustre ce fonctionnement. Attention toutefois, l'utilisation de templates sur un site oblige à prévoir davantage, d'imaginer ce qu'on souhaite obtenir avant de commencer à coder.

1.5. Symfony est orienté objet

Le code est structuré en classes qui permettent d'instancier des objets rendant différents services par le biais de leur méthodes.

1.6. Quelques liens indispensables

- La doc officielle Symfony, disponible en français : http://www.symfony.com/doc/current/index.html
- Le nouveau site du zéro : http://fr.openclassrooms.com/informatique/cours/developpez-votre-site-web-avec-le-framework-symfony2
- La doc Doctrine : http://docs.doctrine-project.org/en/latest/index.html
- La doc Twig : http://twig.sensiolabs.org/

2. INSTALLATION DE SYMFONY

Symfony est livré sous forme d'un dossier compressé. Il suffit juste de décompresser l'archive dans le répertoire de publication de votre serveur web.

- Pré-requis:
  - Wamp server ou équivalent
  - réglages PHP (install PDO et autres extensions php nécessaires...)
- Download de l'archive et décompression dans le répertoire de publication web (exemple : c:/wamp/www)
- Test : http://localhost/symfony/web/app_dev.php doit affiché un message de succès
- Vérifier également que vous pouvez exécuter php en ligne de commande.
  Sous une invite de commande, exécuter: `php -v`
  Le système doit afficher la version de php (au moins 5.4.X.)
  Dans le cas contraire, modifier le path (var d'envir) – clic droit sur ordinateur...

3. ARCHITECTURE DES APPLICATIONS SYMFONY

3.1. L'organisation des fichiers

- **Le répertoire /app** : contient tout ce qui concerne votre site internet… sauf le code source. Il contient donc : la configuration, le cache, les fichiers logs, etc.
- **Le répertoire /src :** contient tout le code source. C'est ici qu'on travaillera la plupart du temps. C'est ici que se trouveront toutes les briques de l'application, c-à-d. tous les bundles.
- **Le répertoire /vendors :** contient toutes les bibliothèques externes à l'application. Dans ces bibliothèques externes, est inclus Symfony2, Doctrine, Twig...
- **Le répertoire /web :**contient tous les fichiers destinés à vos visiteurs (et donc le seul qui devrait à terme être acessible) : images, fichiers CSS et JavaScript, etc.

3.2. Les environnements :

Il existe deux environnements sous symfony : un de test et un de production. Celui de production est plus rapide en chargement mais n'affichera pas les erreurs de développement. Nous travaillerons dans l'environnement de développement.

4. UNE PREMIERE PAGE

4.1. LA NOTION DE BUNDLE

Un bundle est une brique d'une application. Cela consiste donc à regrouper dans un même endroit nommé bundle, tout ce qui concerne une même fonctionnalité.
Un Bundle est, donc, un ensemble de fichiers et répertoires permettant d'implémenter une ou des fonctionnalités (si possible réutilisables dans divers projets). Un Bundle est donc une sorte de package comprenant toutes les couches métiers permettant de répondre à un besoin (grosse fonctionnalité).
La découpe en bundles permet l'échange de bundles entre applications ! Cela signifie que vous pouvez développer une fonctionnalité, puis la partager avec d'autres développeurs ou encore la réutiliser dans un de vos autres projets. Et bien entendu, cela marche dans l'autre sens : vous pouvez installer dans votre projet des bundles qui ont été développés par d'autres !
Exemple :

- Un bundle Utilisateur, qui va gérer les utilisateurs ainsi que les groupes, intégrer des pages d'administration de ces utilisateurs, et des pages classiques comme le formulaire d'inscription, de récupération de mot de passe, etc.
- Un bundle Boutique, qui va fournir des outils pour gérer des produits et des commandes.

4.2. Les bundles de la communauté

Il existe un certain nombre de bundles déjà développés, prêt à l'emploi et donc réutilisables. Presque tous les bundles de la communauté Symfony2 sont regroupés sur un même site : http://knpbundles.com/
Il en existe beaucoup, nous en utiliserons un:

- FOSUserBundle : c'est un bundle destiné à gérer les utilisateurs d'un site. Il fournit le modèle utilisateur ainsi que le contrôleur pour accomplir les actions de base (connexion, inscription, déconnexion, édition d'un utilisateur, etc.) et fournit

aussi les vues qui vont avec. Bref, il suffit d'installer le bundle et de le personnaliser un peu pour obtenir les fonctionnalités de gestion des utilisateurs, de leurs droits !

4.2.1. Structure d'un bundle

Un bundle contient tout : contrôleurs, vues, modèles, classes personnelles, etc. Bref, tout ce qu'il faut pour remplir la fonction du bundle
Exemple : src/Acme/DemoBundle/

```
/Controller       | Contient vos contrôleurs
/DependencyInjection | Contient des informations sur votre bundle (chargement automatique de la configuration p
ar exemple)
/Entity           | Contient vos modèles
/Form             | Contient vos éventuels formulaires
/Resources
-- /config        | Contient les fichiers de configuration de votre bundle (nous placerons les routes ici, par exemple)
-- /public        | Contient les fichiers publics de votre bundle : fichiers CSS et JavaScript, images, etc.
-- /views         | Contient les vues de notre bundle, les templates Twig
/Tests            | Contient vos éventuels tests unitaires et fonctionnels. Nous développerons sans faire de tests au déb
ut.
```

4.2.2. Creation d'un Bundle

Les bundles Symfony doivent être nommés d'une façon précise et doivent suivre la règle de nommage en CamelCase.

```
C:\wamp\www\Symfony>php app/console generate:bundle
```

- Se placer sous c:\wamp\www\Symfony et préciser :
- nom du namespace du bundle = **webStudent/EtudiantBundle**
- nom du bundle = **webStudentEtudiantBundle**
- destination = laisser par défaut c:/w.../www/src
- format de config = **yml**
- structure à générer = **yes** (pour tout générer)
- confirmer = yes

Le système lance la génération
- demande à mettre à jour le kernel = yes
- demande la maj automatique des routes = yes

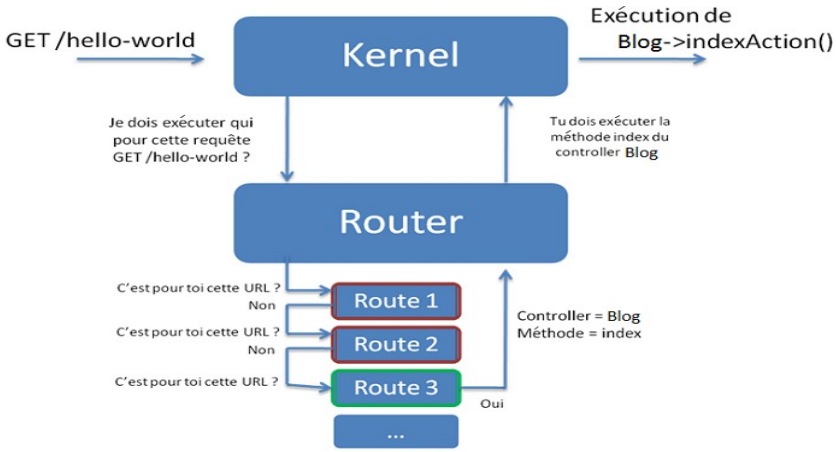
Vérifications :
1. Voir ce qui a été crée dans le répertoire src

2. Un Bundle doit toujours être référencé dans AppKernel.php pour pouvoir être utilisé dans notre projet. Vérifiez que vous avez bien vtre bundle mentionné dans votre fichier app/AppKernel.php:

3. tester http://localhost/Symfony/web/app_dev.php/hello/zakina

4.3. Créer la route

L'objectif du routeur est de dire à Symfony2 ce qu'il doit faire lorsque l'on appelle l'URL /salut-zakina (par exemple). Nous devons donc créer une route qui va dire : « Lorsque l'on est sur l'URL /hello-zakina, alors on appelle le contrôleur "Etudiant" qui va afficher un "Salut Tout le monde, Lebes ? ". »



4.3.1. Création du fichier des routes

La console a crée ce fichier pour nous. C'est un fichier au format **YML**.
`C:\wamp\www\Symfony\src\webStudent\EtudiantBundle\Resources/config/routing.yml`
Ajouter dans ce fichier :

```
L1  SalutEverybody:
L2      path:     /affichSalut
L3      defaults: { _controller: webStudentEtudiantBundle:Etudiant:index }
```

L1 :Nom de la route. A nommé comme vous voulez mais de façon cohérente. Il vous permet juste de vous y retrouver par la suite. La seule contrainte est qu'il soit unique.

L2 : path correspond à l'URL à laquelle nous souhaitons que notre « salut zakina, lebes » soit accessible. C'est ce qui permet à la route de dire : « Cette URL est pour moi, je la prends.
defaults correspond aux paramètres de la route, dont :

`L 3 : _controller`, qui correspond à **l'action** (ici, « index ») que l'on veut exécuter et au **contrôleur** (ici, « Etudiant ») que l'on va appeler (un contrôleur peut contenir plusieurs actions, c'est-à-dire plusieurs pages).

Le nom du contrôleur appelé est composé ainsi :

- « webStudentEtudiantBundle » est le nom de notre bundle, celui dans lequel Symfony2 ira chercher le contrôleur.
- « Etudiant » est le nom du contrôleur à ouvrir. En terme de fichier, cela correspond à `controller/EtudiantController.php` dans le répertoire du bundle. Dans notre cas, nous avons `src/webStudent/EtudiantBundle/controller/EtudiantController.php` comme chemin absolu.
- « index » est le nom de la méthode à exécuter au sein du contrôleur.

!!! Attention en YML 4 espace et surtout pas d'indentation + mode encodage (UTF8 sans BOM)
ni de retour chariot

4.4. Création du contrôleur

Le contrôleur « utilise » tous les autres composants (base de données, formulaires, templates, etc.) pour générer la réponse suite à notre requête ;
C'est ici que résidera toute la logique de votre application.
Symfony en crée un par défaut mais vous allez créer le votre avec ce code ci (fichier EtudiantController.php)

```php
<?php

// src/webStudent/EtudiantBundle/Controller/EtudiantController.php
namespace webStudent\EtudiantBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Response;

class EtudiantController extends Controller
{
       public function indexAction()
        {
              return new Response("Salut tout le monde, lebes ?");
       }
}
```

- **TEST : http://localhost/symfony/web/app_dev.php/affichSalut**

4.5. Création des templates avec Twig

4.5.1. Création du fichier dans le répertoire views :

- creation du repertoire Etudiant et du fichier index.html.twig (html indque que c'est ce qui sera généré)

Conventions de nommage : Etudiant/`index.html.twig` :

- `Etudiant/` est le nom du répertoire. Nous l'avons appelé comme notre contrôleur afin de nous y retrouver (ce n'est pas une obligation, mais c'est fortement recommandé).
- `index` est le nom de notre template qui est aussi le nom de la méthode de notre contrôleur (*idem*, pas obligatoire, mais recommandé).
- `html` correspond au format du contenu de notre template. Ici, nous allons y mettre du code HTML.
- `twig` est le format de notre template. Ici, nous utilisons Twig comme moteur de templates, mais il est toujours possible d'utiliser des templates PHP.

```
{# src/webStudent/EtudiantBundle/Resources/views/Etudiant/index.html.twig #}
<!DOCTYPE html>
<html>
 <head>
 <title>Bienvenue sur votre première page créée avec Symfony !</title>
 </head>
 <body>
 <h1>Salut les étudiants</h1>

 <p>
 Félicitations, vous avez réussi à créer une première page. Vous avez compris ?!?
 </p>
 </body>
</html>
```

4.5.2. Appeler ce template (cette vue) avec notre contrôleur

```
 public function indexAction()
 {
 //return new Response("Salut tout le monde, lebes ?");
 return $this->render('webStudentEtudiantBundle:Etudiant:index.html.twig');
 }
```

4.6. En résumé

- Le rôle du routeur est de déterminer quel route utiliser pour la requête courante.
- Le rôle d'une route est d'associer une URL à une action du contrôleur.
- Le rôle du contrôleur est de retourner au noyau un objet `Response`, qui contient la réponse HTTP à envoyer à l'internaute (page HTML ou redirection).
- Le rôle des vues est de mettre en forme les données que le contrôleur lui donne, afin de former une page HTML, un flux RSS, un e-mail, etc.

5. ACTIVITE

5.1. Dans le bundle déjà créé, créer une seconde page affichant « au revoir tout le monde ». L'Url à appler sera : http://localhost/symfony/web/app_dev.php/auRevoir.

5.2. Créer tous les bundles nécessaires à votre application et dans chaque bundle, développer une page d'accueil personnalisé.