Initiation à Symfony version 3.4

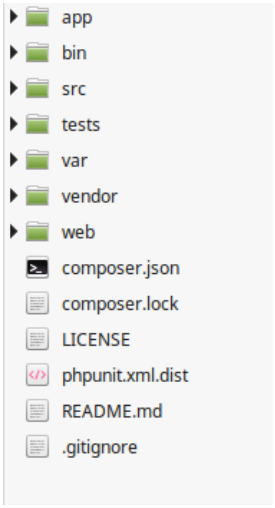
1. Installation d'un projet symfony à partir de composer

Composer (https://getcomposer.org) est l'outil indispensable afin de débuter correctement un projet sous php (même en dehors d'une application développée sous symfony). Il permet,par exemple, via le fichier composer.json de définir les librairies dont votre application aura besoin (comme par exemple le moteur de template twig ou un orm tel que doctrine ou bien propel). Ce fichier permet de configurer les dépendances entre les librairies, la version minimale requise pour php et de paramètrer les répertoires de recherche des définitions de classes. Par défaut, toute nouvelle librairie sera installée dans le répertoire **vendor** à la racine de votre projet.

- Télécharger la denière version du fichier composer.phar à partir de l'adresse du site
- Vérifier dans une console (ou cmd sous windows) qu'une version de php est disponible en tapant la commande **php --version**. Il vous faudra au minimum une version 5.6.
- Si tout est correct, taper la commande :

php composer.phar create-project symfony/framework-standard-edition nomProjet "3.4.*"

- nomProjet représente le nom ainsi que le répertoire de création de votre projet symfony en version 3.4
- Si tout se passe correctement, votre répertoire devrait avoir l'arborescence suivante :



Vous devez retrouver le répertoire **vendor** et le fichier **composer.json.** Le répertoire **web** contiendra tous vos assets (ressources publiques de votre site comme par exemple vos feuilles de style, vos .js vos images). Le répertoire **var** contiendra tous vos fichiers de logs (utilile dans certains cas en mode de développement). Le répertoire **tests** pour effectuer vos tests unitaires. Dans le répertoire **src** vous allez définir tous les controlleurs dont vous aurez besoin pour faire tourner votre application. Le répertoire **app** contient entre autre vos différentes vues développées avec twig et des fichiers de configuration au format yml. Le dernier répertoire **bin** contient l'outil principal de symfony à savoir le script php **console**.

C'est avec ce script que vous allez pouvoir lancer le serveur local en tapant la commande

php console server:start &

Normallement le script vous indique qu'un serveur web est lancé à l'adresse 127.0.0.1:8000. Essayez cette adresse dans un navigateur. Si tout se passe correctement vous devriez voir une page symfony apparaître. En bas de la page, vous devriez voir la barre d'outils de symfony.

- Dans le répertoire src/AppBundle/Controller/, créez le fichier MyController.php et copier le code suivant :

```
<?php
namespace AppBundle\Controller;

use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;

class MyController
{
    public function number()
    {
        $number = mt_rand(0, 100);

        return new Response(
            '<html><body>Tirage aléatoire: '.$number.'</body></html>'
        );
    }
}
```

- Allez ensuite dans le fichier app/config/routing.yml et ajoutez les lignes suivantes :

```
app_tirage:
    path: /nombre/aleatoire
    controller: AppBundle\Controller\MyController::number
```

Attention : le format yml est assez capricieux (tabulation et non pas des espaces pour l'indentation).

Votre seconde route vient d'être créée. Elle porte l'étiquette app_tirage, son chemin d'accès est /nombre/aleatoire et le controlleur qui sera créé par le routeur de symfony sera MyControlleur.php avec l'exécution de la fonction number.

- En mode console, tapez la commande **php console debug:router.** Vous devriez voir apparaître ce nouveau chemin. Testez dans votre navigateur cette nouvelle page (127.0.0.1:8000/nombre/aleatoire). Le routeur lancera automatiquement la méthode number définie dans MyControlleur. Cette méthode renvoit pour le moment un objet Response représentant le résultat à afficher. Il est impossible sous symfony de retourner un résultat au moyen d'un simple echo en php. Il faut soit retourner un template via twig soit l'objet Response.
- A chaque fois que vous définirez une nouvelle action dans votre application, il faudra paramètrer correctement les routes dans le fichier routing.yml ou bien utiliser une manière plus souple : l'utilisation des annotations.

2. Jouer avec les routes sous symfony

Dans le fichier routing.yml vous avez remarqué la présence de ces lignes :

```
app:
    resource: '@AppBundle/Controller/'
    type: annotation
```

Cela indique à symfony d'activer l'utilisation des annotations afin de paramètrer les routes. Dans les différentes actions de vos controlleurs vous allez ajouter un commentaire de la manière suivante :

```
/**
 * @Route("/le/chemin/de/la/route", name="route1")
 */
```

La directive @Route permet de définir la route d'accès à la méthode du contrôleur à partir du navigateur. Chaque route devra être identifiée de manière unique.

- Essayez d'ajouter une nouvelle action dans votre classe MyControlleur qui retournera un objet Response (peut importe le contenu) et vérifiez (après insertion de l'annotation) que la route est accessible dans votre navigateur.

Pour le moment nous n'avons créé que des routes statiques. Il est possible de créer des routes dynamiques sous symfony. La syntaxe de base est la suivante :

```
/**
 * @Route("/le/chemin/de/la/route/{nombre}", name="route2")
 */
```

La fonction qui suit cette annotation devra obligatoirement comporter un paramètre $nombre afin que l'action puisse s'exécuter normalement.

- Ecrire la route /add/{x}/{y} et l'action correspondante dans MyControlleur. Cette action devra effectuer l'addition des deux nombres et retourner le résultat sous la forme « la somme de $x+$y vaut $res » où $x,$y et $res seront remplacées par leurs valeurs.
- Comme les variables ne sont pas typées en php il est tout à fait possible d'exécuter une route du type /add/12/toto. Afin de spécifier un peu mieux les valeurs des paramètres il est possible de rajouter des prérequis dans l'annotation. En regardant l'aide sur symfony dans la section routing, essayer de faire en sorte que les seules routes possibles sont celles avec des valeurs $x et $y de type numérique.

3. Utilisation de twig

- Identifier dans l'arborescence l'endroit où sont stockées les vues et modifier les afin de mettre en place votre propre template web responsive (foundation ou bootstrap) et vérifier que tout fonctionne correctement à l'affichage.

4. Les formulaires

- Ecrire un formulaire contenant les deux zones représentant les variables x et y de la méthode add précédente et faire en sorte d'avoir deux actions avec la même route mais différant sur la méthode d'envois des données (une en GET et l'autre en POST). Lorsque vous cliquez sur le bouton calculer, la bonne méthode devra être appelée.

5. Doctrine

1. Configuration de la base de données

Dans le fichier parameter.yml configurer correctement votre accès à la base de données (soit sqlite soit mysql). Si vous utilisez mysql il faudra créer un utilisateur possédant des droits minimums afin de pouvoir accéder à votre base de données.

La commande *doctrine:database:create* vous permet de créer la base de données (si vous utilisez sqlite, cela créera le fichier représentant votre base de données).

La commande *doctrine:generate:entity* permet de créer une entité en mode interactif.

La commande *doctrine:schema:update* permet de créer la table en relation avec vos entités créées. Pour finir la commande *doctrine:generate:crud* permet de créer un crud (create,read,update,delete) sur une entité passé en paramétre (formulaires, controleur et routes). Pour cela il faudra au minimum avoir les getters et setters d'opérationnels dans vos entités.

Nous n'aborderons pas ici la gestion des relations entre les différentes entités. Pour plus d'informations se reporter à la documentation de doctrine.

6. Installation d'un bundle

Le crud offert par doctrine sur les entités n'étant pas si convivial à utiliser (pas de pagination sur le tableau des entités), nous allons nous intéresser à l'installation d'un bundle un peu plus intéressant.
La page du bundle est disponible à l'adresse suivante :
https://packagist.org/packages/petkopara/crud-generator-bundle

Dans votre fichier composer.json à la racine de votre projet ajouter l'entrée "petkopara/crud-generator-bundle":"3.0.5" dans la partie "require"
Ensuite en ligne de commande à l'aide de composer faites un update (php composer.phar update).
Si tout se passe correctement la librairie devrait s'installer dans le répertoire vendor.
Editer ensuite votre fichier AppKernel.php et ajouter les entrées :

```
new Lexik\Bundle\FormFilterBundle\LexikFormFilterBundle(),
new Petkopara\MultiSearchBundle\PetkoparaMultiSearchBundle(),
new Petkopara\CrudGeneratorBundle\PetkoparaCrudGeneratorBundle(),
```

Vérifier à l'aide de la console de symfony que la commande `petkopara:generate:crud` est disponible.

Le script qui suit devrait vous permettre de générer automatiquement votre backend afin d'alimenter vos tables. Il permet d'une part de supprimer les entités générées du répertoire Entity, le controleur associé à chacune de vos entités du répertoire Controller ainsi que les formulaires pour alimenter vos tables. Ensuite le script exécute une commande qui se connecte à votre base de données et génère vos entités en fonction de vos tables et relations. Pour finir on exécute la commande petkopara:generate:crud sur chaque entité.

Remarque : si vous êtes sous windows il faudra vous inspirer de ce script pour générer correctement toutes vos interfaces.

```
#configurer correctement le fichier app/config/parameters.yml pour PDO
for i in `ls src/AppBundle/Entity/*.php`
do
entite=${i##*/}
entite=${entite%%.php}
echo "suppression de src/AppBundle/Controller/$entiteController.php"
rm -rf "src/AppBundle/Controller/${entite}Controller.php"
done

rm -fr src/AppBundle/Entity
rm -fr src/AppBundle/Form
rm -fr src/AppBundle/Resources

php bin/console doctrine:mapping:import --force AppBundle xml
#création des schemas des tables
#sous la forme de fichiers xml dans le répertoire
# src/AppBundle/Resources/config/doctrine

php bin/console cache:clear

#cette commande permet de générer les entités doctrine en utilisant le système des annotations
# pour le moment cela ne crée que les entités avec les propriétés uniquement
php bin/console doctrine:mapping:convert annotation ./src

#cette commande permet de rajouter les méthodes d'acces
php bin/console doctrine:generate:entities AppBundle

#doctrine ne permet pas de générer d'un seul coup tous les formulaires
#par contre on peut écrire facilement un script sous linux qui permet d'automatiser
#la création de tous les formulaires

for i in `ls src/AppBundle/Entity/*.php`
do
entite=${i##*/}
entite=${entite%%.php}
entite="AppBundle:$entite"
#php bin/console doctrine:generate:crud --entity=${entite} --format=annotation --with-write --no-interaction
php bin/console petkopara:generate:crud -o --entity=${entite} --format=annotation --template=baseBOOT.html.twig --no-interaction
done
```

Ajouter dans le fichier config.yml l'entrée :

templating:
engines: ['twig']

dans la section framework.

7. **Sécurité et Authentification**
   voir la documentation https://symfony.com/doc/3.4/security/entity_provider.html pour pouvoir se connecter et restreindre l'accès de certaines parties du site.

Les points importants :

1. La configuration du fichier security.yml

```
# To get started with security, check out the documentation:
# https://symfony.com/doc/current/security.html
security:
    role_hierarchy:
        ROLE_ADMIN:       ROLE_USER

    encoders:
        AppBundle\Entity\Utilisateur:
            algorithm: plaintext
    # https://symfony.com/doc/current/security.html#b-configuring-how-users-are-loaded
    providers:
        my_provider:
            entity:
                class: AppBundle:Utilisateur
                property: email

    firewalls:
        # disables authentication for assets and the profiler, adapt it according to your needs
        dev:
            pattern: ^/(_(profiler|wdt)|css|images|js)/
            security: false
        main:
            logout:
                path:   /logout
                target: /login

            anonymous: ~

            provider: my_provider

            form_login:
                login_path: login
                check_path: login
    access_control:
        # require ROLE_ADMIN for /admin*
        - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
        - { path: ^/, roles: ROLE_ADMIN }
```

Dans ce fichier vous pouvez paramétrer :

- la hiérarchie des rôles : un rôle administrateur héritera par exemple du rôle user
- l'encodeur utilisé (votre entité que vous utilisez pour rechercher les informations)
- le provider : indique la propriété de votre entité qui jouera le rôle de votre clé primaire
- la partie firewalls : ce qui est important de retenir c'est la section **main** qui contient la partie logout (par défaut l'action est associée à la route /logout le target doit représenter la route de votre page d'accueil par exemple), la partie form_login (la route par défaut est /login et le provider que vous utilisez)

- access_control : c’est ici que vous parametrer vos routes à protéger par un droit spécifique. Il faut écrire les règles par ordre de priorité. Deux rêgles sont définies dans le fichier au dessus : /login qui doit être accessible sans authentification et / (tout le site) qui est protégé par le ROLE_ADMIN

Il faut créer ensuite (ou ajouter des fonctionnalités) à l’entité qui servira à s’authentifier. (la documentation est disponible à https://symfony.com/doc/3.4/security/entity_provider.html)
Un squelette pourrait être celui ci :

```
// src/AppBundle/Entity/User.php
namespace AppBundle\Entity;

use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\UserInterface;

/**
 * @ORM\Table(name="app_users")
 * @ORM\Entity(repositoryClass="AppBundle\Repository\UserRepository")
 */
class User implements UserInterface, \Serializable
{
    public function __construct()
    {
        $this->isActive = true;
        // may not be needed, see section on salt below
        // $this->salt = md5(uniqid('', true));
    }

    public function getUsername()
    {
        return $this->username;
    }

    public function getSalt()
    {
        // you *may* need a real salt depending on your encoder
        // see section on salt below
        return null;
    }

    public function getPassword()
    {
        return $this->password;
    }

    public function getRoles()
    {
        return array('ROLE_USER');
    }

    public function eraseCredentials()
    {
    }

    /** @see \Serializable::serialize() */
    public function serialize()
    {
        return serialize(array(
            $this->id,
            $this->username,
            $this->password,
            // see section on salt below
            // $this->salt,
        ));
    }

    /** @see \Serializable::unserialize() */
    public function unserialize($serialized)
    {
        list (
            $this->id,
            $this->username,
            $this->password,
            // see section on salt below
            // $this->salt
        ) = unserialize($serialized);
    }
}
```

Votre entité doit implémenter les interfaces UserInterface et \Serializable (et donc les méthodes getUsername, getSalt, getPassword, getRoles, erazeCredentials, serialize et unserialize)

La méthode la plus importante étant getRoles qui doit retourner un tableau du rôle (ou des roles) de l'utilisateur qui essaye de se connecter (il faudra écrire correctement cette fonction par rapport à votre modélisation). L'interface \Serializable permettra au service d'authentification de symfony d'engeristrer en session l'utilisateur connecté.

Il reste enfin à mettre en place le formulaire de connexion. Pour ce faire la documentation https://symfony.com/doc/3.4/security/form_login_setup.html devrait vous y aider.

8. Gestion des pages d'erreurs

En mode développement c'est symfony qui intercepte toutes les erreurs http et vous affiche une longue liste de débuggage. Par exemple, lorsqu'une route n'existe pas c'est l'erreur 404 qui est générée. Si vous voulez rediriger vers une page standard la documentation à l'adresse https://symfony.com/doc/3.4/controller/error_pages.html devrait vous y aider. Il faut simplement créer autant de templates que d'erreurs (404, 403, 500, etc.)

9. Conclusions

Vous disposez actuellement de toutes les informations pour démarrer un projet sous symfony qui inclut :

- la création d'un projet vide
- de gérer les routes en utilisant le système d'annotations
- de créer des formulaires de base
- d'installer le bundle Sonata afin de générer le back-end par rapport à votre base de données
- de mettre en place une sécurité pour protéger certaines pages de votre site
- de gérer les exceptions par des pages standard.

Ce qui manque :

- La validation des valeurs des attributs des entités (https://symfony.com/doc/3.4/components/validator.html)
- l'internationalisation (https://symfony.com/doc/3.4/translation.html)
- la gestion des relations many2many dans Sonata (avec et sans attributs dans la table intermédiare)
- ...

10. Installation de Sonata

Installation du bundle via la commande *php composer.phar require sonata-project/admin-bundle*
Activer le bundle dans le fichier AppKernel.php :

```
new Sonata\CoreBundle\SonataCoreBundle(),
new Sonata\BlockBundle\SonataBlockBundle(),
new Knp\Bundle\MenuBundle\KnpMenuBundle(),
new Sonata\AdminBundle\SonataAdminBundle(),
```

Dans le fichier config.yml ajouter les entrées :

```
sonata_block:
    default_contexts: [sonata_page_bundle]
    blocks:
        # enable the SonataAdminBundle block
        sonata.admin.block.admin_list:
            contexts: [admin]
```

Pour l'internationalisation ajouter aussi dans ce fichier l'entrée :

```
# config/packages/framework.yaml
framework:
      translator: { fallbacks: ["%locale%"] }
```

Dans le fichier routing.yml ajouter les entrées :

```
admin:
    resource: '@SonataAdminBundle/Resources/config/routing/sonata_admin.xml'
    prefix: /admin

_sonata_admin:
    resource: .
    type: sonata_admin
    prefix: /admin
```

Taper les commandes *php bin/console cache:clear* et *php bin/console assets:install*

Installer le bundle en tapant la commande *php composer.phar require sonata-project/doctrine-orm-admin-bundle*

Activer le bundle dans le fichier AppKernel.php en ajoutant l'entrée :

```
new Sonata\DoctrineORMAdminBundle\SonataDoctrineORMAdminBundle(),
```

Il faut ensuite adapter le script comme ceci afin d'utiliser correctement la commande sonata:admin:generate :

```
for i in `ls src/AppBundle/Entity/*.php`
do
entite=${i##*/}
entite=${entite%%.php}
echo "suppression de src/AppBundle/Controller/$entiteController.php"
rm -rf "src/AppBundle/Controller/${entite}Controller.php"
done
rm -fr src/AppBundle/Entity
rm -fr src/AppBundle/Admin
rm -fr src/AppBundle/Form
rm -fr src/AppBundle/Resources/config/doctrine

php bin/console doctrine:mapping:import --force AppBundle xml

php bin/console cache:clear

php bin/console doctrine:mapping:convert annotation ./src

php bin/console doctrine:generate:entities AppBundle

for i in `ls src/AppBundle/Entity/*.php`
do
entite=${i##*/}
entite=${entite%%.php}
#entite="AppBundle:$entite"
#php bin/console doctrine:generate:crud --entity=${entite} --format=annotation --with-write --no-interaction
#php bin/console petkopara:generate:crud -o --entity=${entite} --format=annotation --template=baseBOOT.html.twig --no-interaction
php bin/console sonata:admin:generate -n AppBundle/Entity/${entite}
done
```

et ajouter la ligne dans le fichier config.yml :
- { resource: "@AppBundle/Resources/config/services.yml"}
dans la section imports.