Annexe : Aide Symfony

1 Symfony et Controleurs

Symfony est entre autres un framework Web. Il contient un ensemble de ressources qui facilitent la création d'applications Web utilisant le design pattern MVC (Model View Controller).

1.1 Installations de Symfony

Pour commencer à utiliser `Symfony` il faut installer :

- `composer`, le gestionnaire de dépendances de PHP et
- `symfony`, l'utilitaire qui permet de créer de nouveaux projets.

1.2 Analyse du projet de base

Les dossiers et sous-dossiers sont organisés de sorte à ne pas mélanger les choses qui n'ont rien à voir ensemble.
Le dossier `config/` contient les paramètres de l'application. On modifiera ce dossier pour configurer l'application ou changer son comportement par défaut (chargement de modules complémentaires).
Le dossier `src/` contient les sources de notre application. C'est le dossier qui nous intéresse le plus ici.
Le dossier `templates/` contient les fichiers templates (les patrons) TWIG pour faciliter la génération de pages Web.

1.3 Démarrer l'application

L'étape suivante est de démarrer l'application.
Nous utilisons le serveur web interne de PHP plutôt qu'Apache pour le développement.

```
symfony server:start
```

On peut voir le site dans un navigateur, à l'adresse `http://localhost:8000`

1.4 Contrôleur Action et Route

La page visible à l'adresse `http://localhost:8000/` est une page par défaut, elle correspond à une route et à une action dans le contrôleur par défaut.

Un **contrôleur** est la classe principale qui gère un ensemble d'actions.

Une **action** est une méthode de classe du contrôleur, c'est un peu comme un des services possibles proposés par le contrôleur.

Une **route** est la partie qui vient après le nom de domaine dans une URL au sens de HTTP.

Par exemple dans l'URL `https://www.butroanne.com/but1/R209`, la route est `/but1/R209`.
Il faut configurer le contrôleur pour que ses actions soient liées à des routes.

1.5 Gestion d'un contrôleur

Nous allons créer un contrôleur des actions et des routes associées.

Créer un nouveau fichier `src/Controller/HelloController.php` avec le contenu suivant :

```
<?php
namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Annotation\Route;
use Symfony\Component\HttpFoundation\Response;


class HelloController extends AbstractController
{
  /**
   * @Route("/helloRandom")
   */
  public function randomNameAction(): Response
  {
    return new Response(
      "<html><body><h1>Hello " .
        self::generateRandomName() .
        "</h1></body></html>"
    );
  }

  static function generateRandomName(): string
  {
    $noms = [ "Cercle","Cone","Cylindre","Ellipse","Sphère","Pyramide","Rectangle"];
    $couleur = ["Bleu","Vert","Jaune","Violet","Blanc,"Gris" ,"Noir"];
    return $noms[array_rand($noms)]." ".$couleur[array_rand($couleur)];
  }
}
```

1.6 Création d'une vue

On souhaite maintenant générer des pages Web complètes et pas seulement une chaîne de caractères passée dans l'objet **Response**. Symfony utilise le langage de template twig.

La méthode `render()` du contrôleur permet d'utiliser un template twig pour fabriquer une réponse et retourner un objet `Response`.

Par exemple, représentons un template pour le contrôleur `Hello` dans un fichier `templates/hello.html.twig` :

```
{% extends 'base.html.twig' %}
{% block title %}Hello!{% endblock  %}
{% block body %}
    <div id="container">
        <div id="hello">
            <h1> Hello <i>{{ name }}</i>!</h1>
        </div>

    </div>
{% endblock %}
{% block stylesheets %}
<style>
    body { background: #F5F5F5; font: 18px/1.5 sans-serif; }
    h1, h2 { line-height: 1.2; margin: 0 0 .5em; color: #4343AA;}
    h1 { font-size: 36px; }

    }
</style>
{% endblock %}
```

2 Symfony et Bases de Données

2.1 Installation et Configuration

Symfony seul ne permet pas de gérer des modèles objets ni de se connecter à une base de données.
On utilise **Doctrine**, un ORM (object-relational mapping) pour mettre en concordance des objets PHP avec un modèle persistant (Base de données).

Création de la base de données :

```
php bin/console doctrine:database:create
```

Tout effacer et recréer la basse de données (Attention !) :

```
php bin/console doctrine:database:drop --force

php bin/console doctrine:database:create
```

2.2 Création d'une entité

Une entité représente le type d'objets auxquels on s'intéresse dans l'application. C'est avant tout une classe PHP.
Par exemple, prenons l'exemple développé dans le document Databases and Doctrine sur le site de Symfony.
Par convention on crée les modèles dans le sous-dossier `Entity`.
Nous pouvons utiliser l'utilitaire de création d'entités :

```
symfony console make:entity
```

Ce script est interactif et nous permet de définir une entité avec ses champs.
Ce script produit 2 fichiers :

- `src/Entity/Product.php`
- `src/Repository/ProductRepository.php`

Dans `Product.php` :

- par défaut la table `product` est liée au modèle objet `Product` (c'est modifiable)
- un champ `id` a été ajouté, c'est la clé primaire
- les noms des colonnes portent le nom des champs (c'est modifiable)
- les `setters` retournent l'objet courant. On peut faire du chainage de méthodes

2.3 Persistance du modèle objet

Une fois le modèle défini, on peut générer la table associée dans la base de données. On appelle cela la migration.

```
symfony console make:migration
```

Cette commande est très puissante, elle compare les tables et les modèles existants et génère le code SQL approprié. (e.g. utilise des "ALTER TABLE" lors de la mise a jour de modèles existants).
Le code php généré pour modifier la base de donnée se trouve dans le dossier `migrations/`'
Pour exécuter la requête et effectivement migrer la base :

```
symfony console doctrine:migrations:migrate
```

2.4 Création et persistance d'objets

On a maintenant un modèle objet persistant opérationnel.
On peut créer, afficher, modifier et supprimer des objets de ce type.
Ce genre d'action se fait naturellement dans un contrôleur.
On utilise la commande suivante pour générer un contrôleur de base :

```
symfony console make:controller ProductController
```

On note :

- l'accès au "gestionnaire d'entités" (`entity manager`) via : `$this->getDoctrine()->getManager()`
- c'est l'objet qui fait réellement les requêtes
- la méthode `persist` qui indique à l'entity manager que l'objet passé en paramètre doit être persisté
- la méthode `flush` exécute toutes les requêtes nécessaires en un seul `prepare`.
- on peut faire plusieurs `persist` pour un seul flush.

Par exemple,

```
<?php

namespace App\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Annotation\Route;
use Doctrine\Persistence\ManagerRegistry;
use App\Entity\Product;

class ProductController extends AbstractController
{

    #[Route('/product/create', name: 'create_product')]
    public function create_product(ManagerRegistry $doctrine): Response
    {
        $entityManager = $doctrine->getManager();

        $product = new Product();
        $product->setName('Keyboard');
        $product->setPrice(1999);
        $product->setDescription('Ergonomic and stylish!');

        // tell Doctrine you want to (eventually) save the Product (no queries yet)
        $entityManager->persist($product);

        // actually executes the queries (i.e. the INSERT query)
        $entityManager->flush();

        return new Response('Saved new product with id '.$product->getId());
    }

    #[Route('/product', name: 'product')]
    public function index(): Response
    {
        return $this->render('product/index.html.twig', [
            'controller_name' => 'ProductController',
        ]);
    }
}
```

On appelle cette route : https ://localhost :8000/product/create

On vérifie l'existence de l'objet dans la base :

```
php bin/console doctrine:query:sql 'SELECT * FROM product'
```

2.5 Consulter un objet

Si l'on connait l'identifiant d'un objet alors il est très simple de le consulter (read) à partir du contrôleur.

On effectue toujours les requêtes de consultation sur un type d'objets grâce à son `Repository` (RepositoryProduct) :

```
$this->getDoctrine()->getRepository('AppBundle:Product')
```

AppBundle :Product est équivalent a `AppBundle\Entity\Product`

2.6 Le Repository

Le `Repository` contient une quantité de méthodes qui facilitent l'accès au modèle de données.

On peut accéder aux objets :

- un par un par leur `id` : `find($id)`
- dynamiquement en fonction des champs : `findOneByName('foo')`, `findOneByPrice(19.99)`
- plusieurs à la fois : `findByPrice(19.99)`
- tous : `findAll()`
- une fonction de plusieurs conditions :

```
// query for one product matching by name and price
$product = $repository->findOneBy(
    array('name' => 'foo', 'price' => 19.99)
);
```

- plusieurs fonctions de plusieurs conditions :

```
// query for all products matching the name, ordered by price
$products = $repository->findBy(
    array('name' => 'foo'),
    array('price' => 'ASC')
);
```

- Jointure entre deux tables :

```
$product = $doctrine->getRepository(Product::class)->findOneByIdJoinedToCategory($id);
```