

1) INTRODUCTION

Symfony 2, en 2011, est le premier framework un peu moderne de PHP. Il a une approche façon boîte à outils, un peu moins monolithique que ses concurrents. Il est composé de différents composants (comme Twig, le router, les formulaires etc) qui liées ensemble, permettent de créer une applications web de manière plus rapide et de manière « standardisée » en forçant les bonnes pratiques aux maximum.

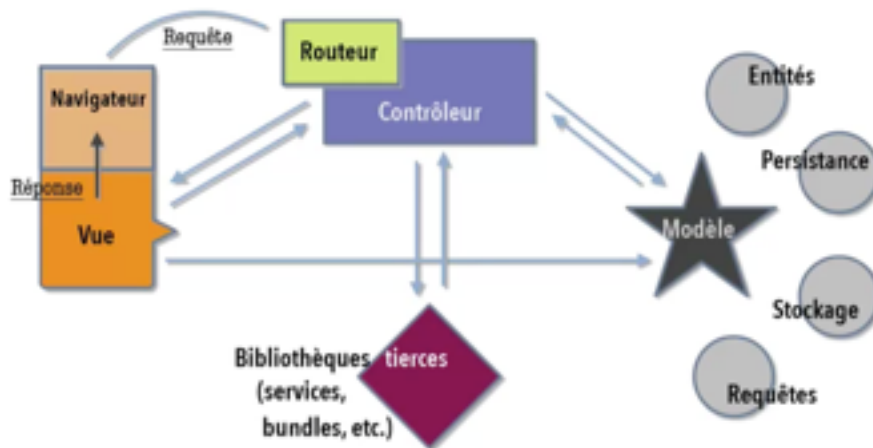
La version 4 pousse encore plus cette approche de boite à outils avec une installation de départ d'un squelette Symfony, presque vide de tout composant, que l'on peut ensuite ajouter au fur et à mesure si besoin.

Aujourd'hui de nombreuses applications utilisent des parties de cette boite à outils : Drupal, Laravel, Prestashop, Magento, PHPBB.

1) Pique de rappel (en bref) :

- PHP 5.3 a introduit les espaces de nom, alias Namespace, pour éviter les collisions de noms. On peut aussi définir des alias / raccourcis de classe pour améliorer la lisibilité. On le définit dans la première ligne PHP et on respecte l'arborescence des dossiers. L'autoload permet de venir charger automatiquement ces classes.
- PSR : C'est une initiative pour normaliser le code PHP avec des recommandations, des conventions, mis en place par le FIG (Framework Interoperability Group). La plus importante est le PSR-4, elle précise comment se fait l'autoloading des classes. Mais il y a de nombreuses recommandations.
- MVC : architecture de code permettant de séparer et organiser les fichiers d'une applications en fonction de leur rôle.
 - Contrôleur : reçoit les requêtes. Chargé de gérer les interactions entre l'utilisateur et le modèle. C'est le chef d'orchestre de l'application.
 - Modele : gère la manipulation et le traitement des données.
 - Vue : présente les données du modèle à l'interface utilisateur. Enregistre les actions utilisateurs (envoi d'un formulaire etc, et les transmet au contrôleur.

2) Cycle de vie d'une requête HTTP dans une application Symfony :



3) Installation de Symfony

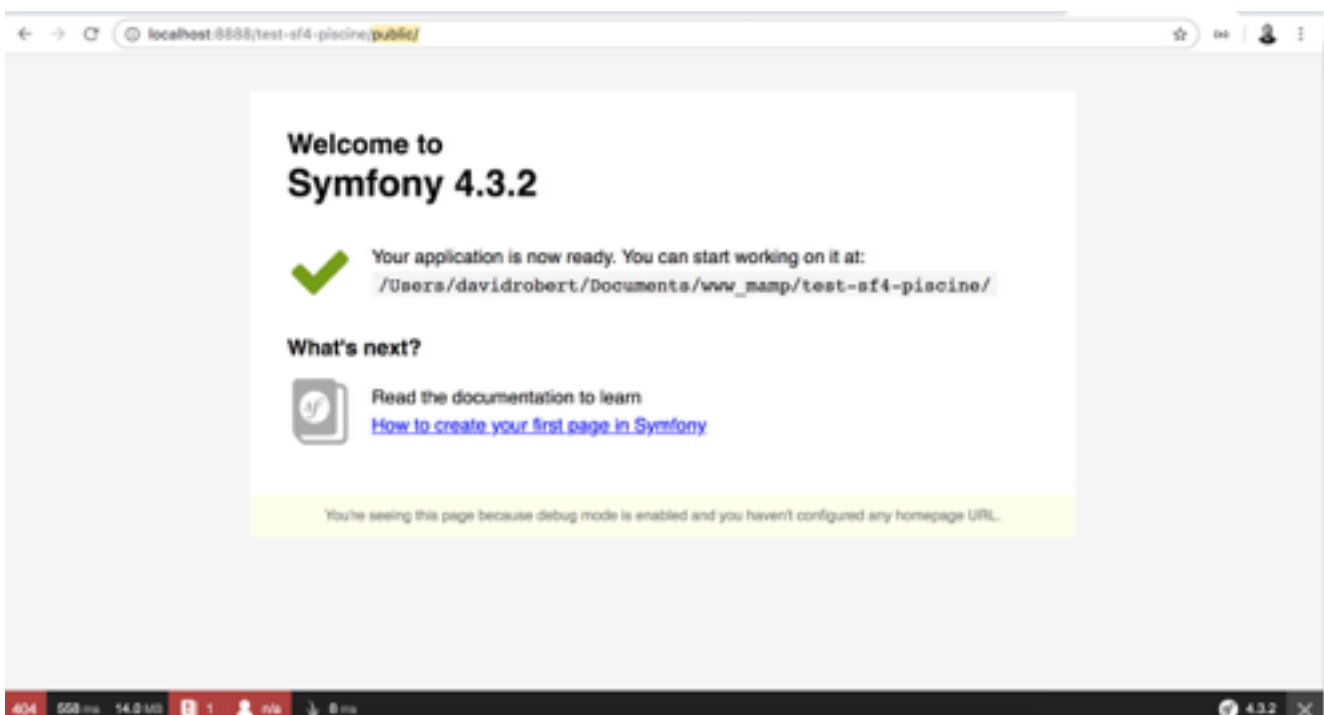
En ligne de commande :

```
composer create-project symfony/website-skeleton nomduprojet
```

Si Apache est utilisé (notamment avec MAMP), il faut configurer le .htaccess. Pour ça on va utiliser :

```
composer require symfony/apache-pack
```

Une fois installé, un message de bienvenu devrait s'afficher en allant sur votre projet via votre serveur local, suivi de « public » :



3) Architecture

- Le dossier « public » est le seul dossier accessible par le navigateur. Il contient un fichier `index.php` qui est le point d'entrée de l'application.
- Le dossier `config`, contient les fichiers de configuration de chacun des packages de l'application.
- Le dossier `src` contient votre code
- Le dossier `templates` contient les fichiers Twig (la partie « vue » de votre application)
- Le dossier `var` contient les logs et les caches
- Le dossier `tests` contient les tests écrits pour votre application
- Le dossier `bin` contient le fichier d'entrée pour la ligne de commande
- Le dossier `vendor` contient les bibliothèques
- Le dossier `translations` contient les traductions de vos pages (si votre applications est en plusieurs langues)
- Le fichier `.env` contient la configuration pour l'accès à la base de données et d'autres informations sens

Environnements disponibles dans Symfony :

- `prod` : le site visible par les utilisateurs. N'affiche pas les erreurs et utilise des fichiers en cache pour optimiser les performances.
- `dev` : pour développer. Permet notamment d'afficher les erreurs et la barre de debug.
- `test` : utilisé pour les jeux de tests.

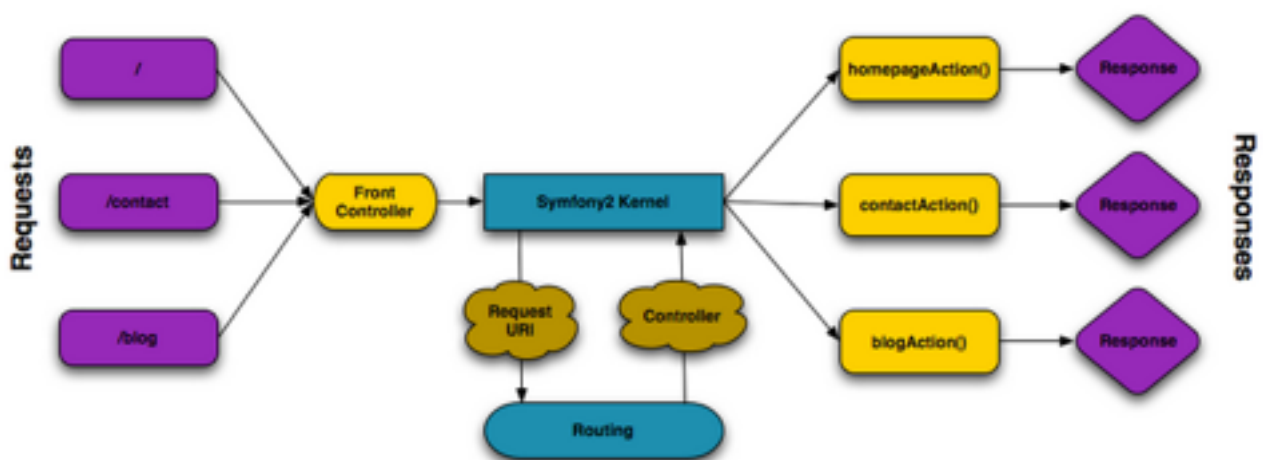
Chaque environnement peut avoir une configuration différente (via des fichiers de configurations spécifiques dans le dossier `config`).

2) LES ROUTES

Les routes dans Symfony permettent de définir une URL, d'identifier une ressource.

Une URL se compose de plusieurs éléments :

- protocole : http
- nom de domaine : www.example.com
- un port : le serveur écoute sur un port
- un chemin : spécifie le contenu auquel on accède via une arborescence
- un nom de ressource : index.html, image.jpg etc
- des options : ?nom=david&prenom=robert



Le composant routing dans symfony :

- gère les collections de routes
- détermine les contextes de requêtes (analyse l'url pour savoir le chemin demandé etc)
- relie une url avec une route

Les routes sont un peu l'API de notre appli. A chaque route fait référence à une méthode de controller.

Une route contient deux propriétés principales :

- id : un identifiant unique qui permet à l'appli d'accéder à la route.
- path : le schéma de l'url qui permettra d'analyser la requête du navigateur.

Les routes s'écrivent en « annotations » au dessus des méthodes de contrôleurs (il est possible de les écrire aussi dans un fichier Yaml, XML ou même PHP). Une annotation est un commentaire PHP qui permet d'exécuter du code.

Pour installer le composant Annotations :

composer require annotations

TP : créer deux routes, qui viennent afficher deux var_dump différents en utilisant ce squelette de controleur :

```
<?php

namespace App\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class HomeController extends Controller
{
    public function index()
    {
        dump('hello'); die;
    }
}
```

3) CONTROLEURS

Une classe controleur dans Symfony doit hériter de la classe `Symfony\Bundle\FrameworkBundle\Controller\AbstractController`.

Le rôle du contrôleur est de recueillir une requête http (filtrée par le routeur), d'effectuer un traitement (récupérations de données dans la bdd, etc) puis de rendre du contenu via la view.

Chaque controleur doit retourner une réponse compatible avec le format http. Pour cela; il faut retourner une instance de la classe `Response` du composant `HttpFoundation`.

Objet Request

La classe `request` = permet de gérer tous les paramètres gérés habituellement par PHP dans les variables super-globales (`$_GET`, `$_POST` etc).

Pour l'utiliser, on passe l'objet en argument de l'action controleur (avec un type hinting vers l'interface ou la classe de `Request`, qui nous permet d'éviter d'avoir à instancier l'objet `Request` à la main).

Et on peut venir récupérer, avec les getters de la classe `Request`, tout ce qu'on veut lié à la requête : `getHost`, `getMethod`, `getClientIp`, `getScheme` (`http`, `https` etc), la langue, le chemin etc.

TP : créer une nouvelle route, et dans un controleur récupérer et afficher l'url + le protocole utilisé pour la requete.

Pour récupérer des parametres GET, on utilise `$request->query->get('nomdu parametre');`

TP : créer une nouvelle route avec des paramètres GET, un chiffre et un bool, et dans un controleur faire une condition en fonction du bool pour afficher le nombre.

Quand on définit une route, on peut ajouter dans le chemin choisi des variables (nommées wildcards) :

Exemple :

```
/**  
  
 * @Route(«/blog/{id} », name=« blog_article »)  
  
 */
```

Exemple d'url :

```
/articles/1  
/articles/2
```

Pour récupérer ces « wildcards », il suffit de passer en argument du contrôleur une variable du même nom que la wildcard, donc ici « \$id »

On pourrait les rendre optionnelles via une entrée default dans nos routes. On pourrait aussi mettre des contraintes sur ces variables pour qu'elles respectent un certain schéma, via une entrée « requirement » dans nos routes.

TP : créer une route avec des variables \$categorie et \$titre, et les afficher dans du html en les ajoutant via le contrôleur avec une réponse.

TP : créer une route avec des variables âge, \$nom \$prenom et faire une condition si l'âge est inférieur à 18 ans afficher un message de refus, sinon afficher le nom et prénom.