CAMPUS ID

COURS PHP - LE FRAMEWORK SYMFONY

Jean-Michel Notteghem

- Développeur Symfony
- Mission chez Orange
- Anciennes expériences en Ruby

Première partie : Symfony en théorie

Nous allons voir ce qu’est Symfony, pourquoi ce projet existe, à quelle problématique tente-t-il de répondre.

Symfony, qu'est ce que c'est ?

Symfony est un framework (cadriciel en français), en PHP. Il s'articule autour d'une architecture MVC (Modèle Vue Controller).

Il est constitué d'un ensemble de composants, parfois gérés par Sensiolabs (l'entreprise qui gère Symfony), parfois gérés par la communauté.

L'objectif de Symfony est de fournir un outil afin de faciliter la réalisation de sites web dynamiques.

Ses concurrents sont nombreux : Laravel, CodeIgniter, Phalcon, Slim,

Laravel, son principal concurrent, utilise une grande partie des composants fournis par Symfony.

Pour info

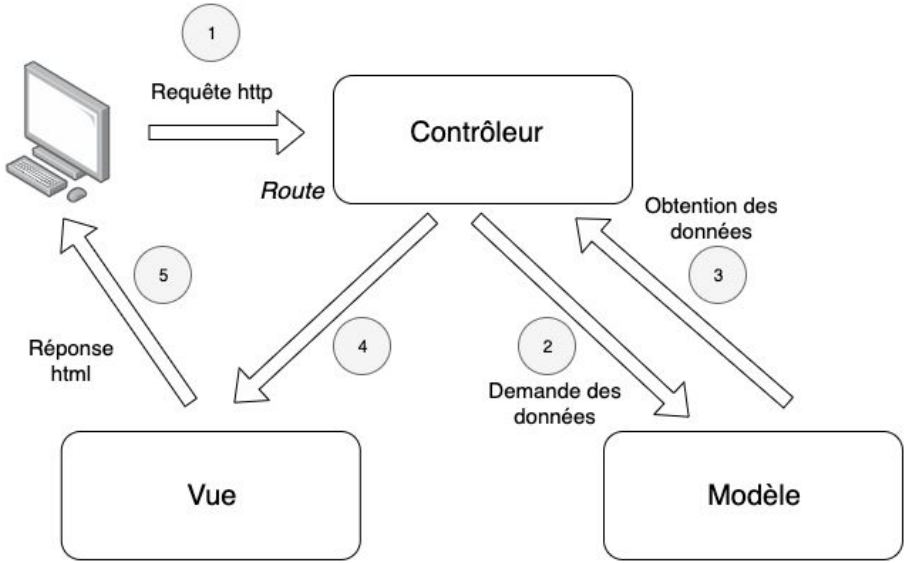
On peut exploiter Symfony afin plus qu'un simple site web dynamique.

Symfony permet de créer des **démons** (application qui tournent en tâche de fond), des **API (RestFul),** voir même des **librairies** complètes à intégrer dans d'autres projets Symfony (ce que l'on appelle des **Bundles**).

Structure MVC

Modèle **V**ue **C**ontrôleur est une architecture logicielle. C'est un concept très souvent utilisé, notamment dans les grand frameworks web.
Le principe consiste à séparer tout ce qui est

- affichage HTML dans une partie : la **Vue**
- traitement de la requête HTTP, ses paramètres, l'url, dans une autre partie : le **Contrôleur**
- les données et leurs gestion dans une dernière partie : le **Modèle**



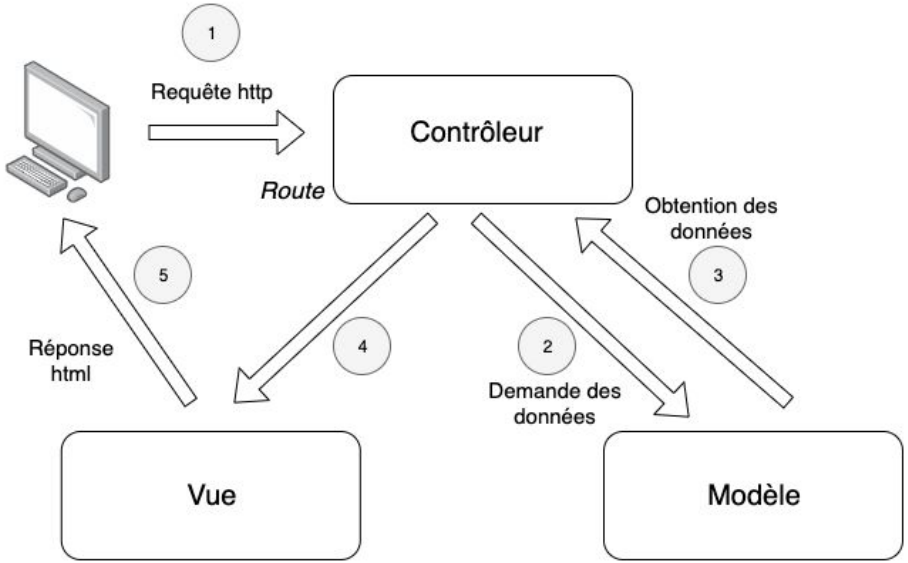
Structure MVC

Les composants au sein de Symfony qui remplissent ces roles sont :

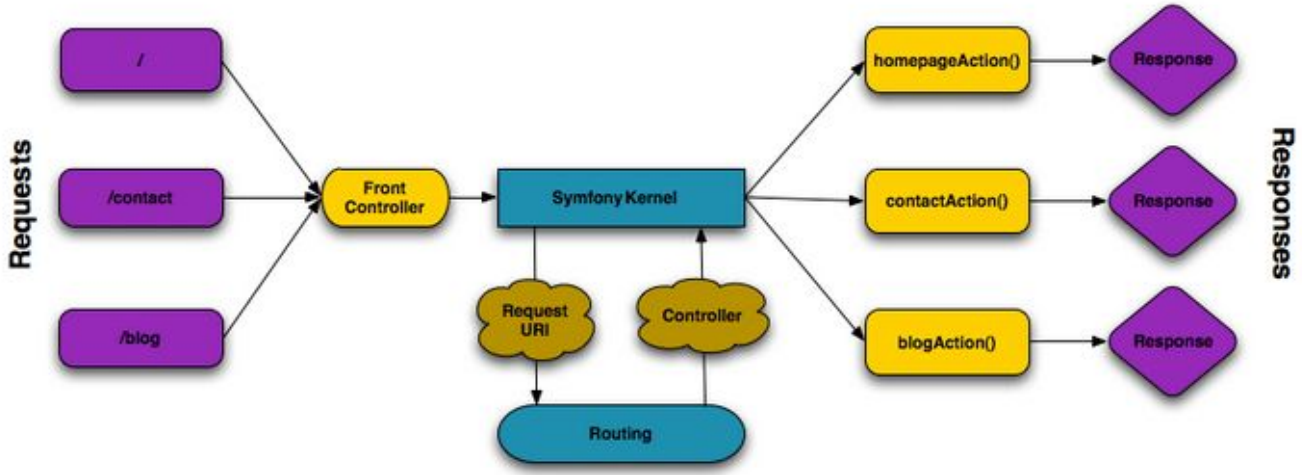
Les Controllers du **FrameworkBundle**.

Les modèles géré à l'aide de **Doctrine**.

Le moteur de template **Twig** pour générer les vues.



Architecture d’appels sous Symfony



https://dev.to/brpaz/an-introduction-to-symfony--the-foundation-of-modern-php-applications-ehj

Les plus de Symfony

Voici les principaux outils que Symfony propose pour aider le développeur :

- Une **interface console** pour automatiser de la génération de code et faire des opérations de maintenance
- Une **barre de débogage** dans le navigateur qui aide à inspecter finement les requêtes, les réponses, les logs applicatifs, le debug, les requêtes SQL lancées… etc.
- Un **serveur web** embarqué

Les plus de Symfony

Niveau organisationnel, voici un aperçu des avantages :

- Une arborescence des fichiers optimisée
- Un système de routing pour bien gérer les urls de votre application
- Une couche ORM (Object Relational Mapping) pour gérer les éléments en base de données plus aisément
- Une génération des formulaires HTML automatisé
- Un système d'authentification assisté
- Un langage de templating pour les vues
- Une intégration avec une chaîne de build frontend
- Une gestion intégrée des variables d'environnement

Seconde partie : Symfony en pratique

Place à la pratique ! Nous allons, pas à pas, installer Symfony, générer un nouveau projet avec, le configurer, et intégrer quelques fonctionnalités afin de voir l'éventails des concepts abordés par Symfony.

Environnement du TP

Ce TP sera organisé au sein d'un environnement Docker.

Je vous fournirais un **docker-compose.yml** ainsi qu'un **Dockerfile** afin de monter 2 images :

- Une image avec PHP8
- Une image avec MySQL

Ainsi, vous verrez l'installation et la configuration de Symfony sous Linux, ce qui est généralement l'environnement d'entreprise. Docker permet aussi de monter les toutes dernières versions de chaque couche logicielle.

Installation

Pour installer Symfony, il faut passer par un script d'installation.
Sous Windows, il se trouve ici sous forme d'exécutable :
https://get.symfony.com/cli/setup.exe

Sous Linux, il suffit d'avoir **wget** d'installé et de lancer une commande bash :

```
wget https://get.symfony.com/cli/installer -O - | bash
```

Wget va télécharger le dernier script d'installation en date, et le "**| bash**" va évaluer le script téléchargé.

Installation

Il faut ensuite se placer dans le dossier /home du conteneur.

Pour générer un nouveau projet, un appel à l'exécutable symfony :

```
symfony new --full [NOMDUPROJET]
```

- symfony : binaire du framework
- new : action pour créer un nouveau projet
- --full : paramètre pour que symfony génère un projet web complet
  Sans ce paramètre, symfony génèrera un projet minimal, pour par exemple une API ou un démon.

Arborescence

Ci contre, voici une arborescence classique d'un projet Symfony.

bin
config
migrations
public
src
src/Controller
src/Entity
src/Form
src/Repository
Kerner.php
templates
tests
translations
var
vendor
.env + .env.test
composer.json + composer.lock + symfony.lock
phpunit.xml.dist

DotEnv

Le fichier “**.env**” contient l’ensemble des variables d’environnement inséré par Symfony.

Lorsque vous souhaitez utiliser Symfony avec une base de données, il faut renseigner la variable DATABASE_URL.

Le format attendu est
“**mysql://**[USER]**@**[HOST]**:**[PORT]**/**[NOM_APP]**?serverVersion=8.0**”;

Dans un fichier “**.env**”, il faut séparer le nom de la variable et sa valeur par un “**=**”;

VARIABLE=VALEUR

Serveur de développement

Symfony propose un serveur web local, afin de travailler sur le projet Symfony sans avoir à gérer un serveur de production, comme Apache ou Nginx.
Le serveur est basé sur le serveur local PHP (https://www.php.net/manual/fr/features.commandline.webserver.php).

Symfony ajoute une gestion des certificats, afin de pouvoir travailler en HTTPS :

```
symfony server:ca:install
```

Ensuite, on peut démarrer le serveur avec cette commande :

```
symfony serve
```

Ajouter le “**-d**” permet de lancer le serveur en détaché (tâche de fond).

```
symfony serve -d
```

Pour info

Après avoir démarré le serveur, on constate que Symfony le fait tourner sur le port 8000 par défaut.

Si vous faites tourner votre Symfony dans un docker, prenez bien garde au mapping de ports.

Ici, on précise le port du Docker qui map le 8000, ou le 8000 directement si vous n'êtes pas sous Docker.

Sur votre navigateur vous arrivez sur la page par défaut de Symfony.

Dans la Web Debug Bar, on peut consulter dans l'ordre :

- Le code retour HTTP
- Le temps de réponse
- La consommation mémoire PHP
- Le nombre de logs d'erreur
- L'état de connexion de l'utilisateur courant
- Le temps de rendu HTML

Dans la Web Debug Bar, est indiqué la version courant du framework.

Notre premier contrôleur

L'application, mis à part la page par défaut, ne fait encore rien. Pour commencer à développer, il nous faut une première route, et donc un premier **contrôleur**.

```
symfony console make:controller IndexController
```

Le fichier sera généré dans /src/Controller, et une route correspond à ceci

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

Pour info

Après avoir généré un **controller** via la commande Symfony, une vue (un template Twig) est généré simultanément.

Twig

Twig est un composant de Symfony (géré indépendamment, mais toujours par SensioLabs, l'entreprise derrière Symfony).

Pour résumer, Twig est un moteur de template (comme **ERB** pour Rails, **Moustache** en JS, **Smarty** en PHP… etc.)

Ses fichiers sont d'extensions "**.twig**" et il comprend une syntaxe dédiée :

{% %} => permet d'interpréter des opérateurs : if, else, for… etc. On y ferme aussi ces opérateurs avec **{% endif %} {% endfor %}**... etc.

{{ }} => permet d'interpréter les variables.

| => permet, à la suite d'une variable, d'y appliquer une fonction (qu'on appelle un filtre twig)

User et autres entités

Pour gérer une authentification sur votre site web, Symfony fournit un “assistant” permettant de mettre ça en place de manière quasi automatique.

```
symfony console make:user
```

L'assistant va créer un modèle User, prêt pour être lié à un système d’authentification.

Autres entités

Maintenant que nous avons un utilisateur en base, ajoutons un autre modèle, appelé ici entité (une entité est un modèle lié à la BD via une couche ORM, ici **Doctrine**.

```
symfony console make:entity
```

L'entité ciblée par la commande peut être une nouvelle entité, mais aussi une entité existante qu'il conviendrait de mettre à jour.

Doctrine gère la couche modèle liée à la base de données. C'est un composant open source intégré à Symfony. Il n'est pas exclusif à Symfony, on le rencontre dans d'autres framework, et il peut être utilisé seul dans un projet PHP natif.

Couche 1 : ORM

Object Relational Mapping, permet de faire le lien entre le modèle relationnel, et un objet PHP.

Couche 2 : DBAL

Database Abstraction Layer, permet de gérer le lien entre le modèle, et le langage de la base de données.

Couche 3 : PDO

Il s'agit de la couche qui gère la connexion technique à la base de données, et la communication des requêtes fournies par la couche DBAL

Migrations Doctrine

Doctrine gère les évolution de schéma de base de données avec des fichiers de migrations. Pour résumer, ces fichiers contiennent les requêtes pour faire évoluer les tables de la base en fonction du modèle des entités. Pour créer une migration après avoir travaillé sur un modèle, il faut exécuter cette commande :

```
symfony console make:migration
```

Dans un fichier migration, il y a le code pour faire la mise à jour du schéma de BD, mais également le code pour le défaire.

Une fois les migrations préparées, pour les appliquer en base de données, il faut exécuter cette commande ci :

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

User et autres entités

Faisons un rapide retour sur notre User. Maintenant qu'il existe en base de données, et qu'il est structurellement prêt à être intégré dans une logique d'authentification, il convient de préparer les mécanisme interactifs qui mettent ceci en oeuvre.

Deux commandes pour préparer ça :

- Créer un formulaire de login

```
symfony console make:auth
```

- Créer un formulaire d'inscription

```
symfony console make:registration-form
```

Pour info

Les formulaires sous Symfony sont géré via des objets “FormType”. Ainsi, lorsque l’on fait évoluer un modèle d’entité, le formulaire est à mettre à jour dans sa définition de classe, et tous les formulaires appelés dans les templates seront mis à jour.

Ce n’est pas forcément tout automatisé, mais cela apporte une grande cohérence au code.

Création de formulaires

Pour créer un formulaire, il faut exécuter cette commande et lui fournir le nom de l'entité que le formulaire concerne :

```
symfony console make:form
```

Cela va créer un nouveau fichier **/form/*Type.php**

La fonction buildForm permet de préparer les champs du formulaire.

```php
public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('name')
            ->add('user')
            ->add('tag')
        ;
    }
```

Intégration de formulaires

Dans une action d'un contrôleur, il convient de créer un formulaire. Pour cela, il faudra avoir une entité du bon type, et une méthode du contrôleur permettra de générer le formulaire correspondant à l'entité :

```php
$item = new Item();
$form = $this->createForm(ItemType::class, $item);

return $this->render('item/new.html.twig', [
    'itemForm' => $form->createView(),
]);
```

Côté vue (en twig donc), on affiche le formulaire à l'aide d'une série de fonctions :

- form_start pour le débuter
- form_row pour ajouter un couple input / label
- form_end pour le terminer

```twig
{{ form_start(itemForm) }}
        {{ form_row(itemForm.name) }}
        <button type="submit" class="btn">Add Item</button>
    {{ form_end(itemForm) }}
```

Soumission de formulaire

Voici le cas classique d'une soumission de formulaire. On part sur la base de notre ancien code de préparation de formulaire au sein du contrôleur, cependant on y ajoute la gestion de la requête.

Du form on appel handleRequest, afin de lier la requête HTTP et le formulaire. Ainsi, les données en POST ont été intégrées. La méthode getData() permet de retourner l'entité populée.

L'objet $request comme $em ont été autowirés, c'est à dire une dépendance injectée via un type hint dans la signature de la function.

```php
$item = new Item();
$form = $this->createForm(ItemType::class, $item);
$form->handleRequest($request);

if($form->isSubmitted() && $form->isValid()) {
    $item = $form->getData();
    $em->persist($item);
    $em->flush();
    return $this->redirectToRoute('items');
} else {
    return $this->render('item/new.html.twig', [
        'itemForm' => $form->createView(),
    ]);
}
```

Autowiring

L'autowiring est un système d'injection de dépendance automatique. Il permet d'exposer des services différents moment dans le code. C'est une technique utilisée dans bien des framework (par exemple dans Spring en Java).

Pour autowirer un service, on passe par un type hint (préfixer l'argument d'une fonction du nom de la classe que l'on souhaite). Symfony s'occupe du reste, grâce à la capacité de réflexion du langage PHP.

Service : au sein de Symfony, on appelle "service" un singleton, injectable, qui sert à mutualiser certains traitements.

EntityManager

L'entity manager est un service fourni par Doctrine qui permet de lier la manipulation des entités au sein d'un contrôleur ou d'un autre service, avec la base de données.

On récupère l'entity manager par autowiring, pour ensuite persister les entités voulues à l'aide de la fonction `EntityManager::persist(Entity $entity)`.

Le traitement se fait en 2 fois. D'abord on persiste toutes les entités que l'on a modifié, pour finalement écrire en une fois dans la base de données à l'aide de la fonction `EntityManager::flush()`.

Arborescence

Rappel de l'arborescence, qui à ce stade du cours, doit avoir bien plus de sens :

bin
config
migrations
public
src
src/Controller
src/Entity
src/Form
src/Repository
Kerner.php
templates
tests
translations
var
vendor
.env + .env.test
composer.json + composer.lock + symfony.lock
phpunit.xml.dist

Pour info

Symfony peut produire des sites web dynamiques, cependant, pour aller plus loin, je vous invite à voir deux bundles Symfony intéressants :

Un bundle pour gérer un démon applicatif :
https://github.com/M6Web/DaemonBundle

Un bundle pour gérer une API Rest :
~~FOSRestBundle~~ ce bundle n'existe plus. Aujourd'hui il est conseillé de générer une appli Symfony minimale à la place.

Merci à tous !

N’hésitez pas à me poser des questions.

jean-michel.notteghem@outlook.com
+336 62 76 94 79