

Symfony Partie 1

Présentation globale

Ce Framework date de 2005. Il est issu de l'agence web française Sensio.

Symfony utilise le modèle MVC. Il sera très difficile de comprendre Symfony si vous n'avez pas bien assimilé le modèle MVC.

Il utilise Doctrine pour la partie modèle. Doctrine est un **ORM** (Object-Relational Mapping). Il réalise donc du mapping objet-relationnel, c'est-à-dire qu'il fait la liaison entre des objets, ici en PHP, vers des données enregistrées dans un SGBD relationnel. Il est indépendant du Framework Symfony et est utilisé dans bien d'autres Framework.

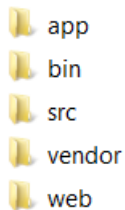
La vue est gérée par un moteur de template, Twig, qui a été réalisé par le même concepteur que Symfony. Ce moteur est toutefois indépendant. Un moteur de template s'occupe de générer l'affichage avec un code particulier. Le plus célèbre reste Smarty.

Twig utilise, par exemple, des balises spécifiques comme {% %} pour des conditions et {{}} pour l'affichage. Il va séparer le code HTML-CSS-JS du PHP.

Architecture

Les répertoires

Nom



- **App** contient les fichiers de configuration et de cache de l'application
- **Bin**
- **Src** contient le code source
- **Vendor** contient les bibliothèques utilisées ou à ajouter, par exemple Doctrine
- **Web** contient les images, le CSS, le JS et le contrôleur principal.

Pour accéder aux applications, deux contrôleurs principaux sont disponibles, app_dev.php et app.php. Le second n'affichera pas les erreurs et est à utiliser en production. On utilisera donc **app_dev.php**.

Les bundles

Toute brique d'une application est appelée **Bundle**. Un bundle effectue une tâche spécifique, complète. On séparera par exemple, la gestion des utilisateurs et la partie administrateur dans un autre bundle.

Tous les fichiers nécessaires à une utilisation spécifique seront regroupés dans un bundle.

On peut télécharger des bundles sur

knpbundles.com

Création d'un projet

On doit passer en premier par la création d'un bundle.

Sous PowerShell (ou le terminal sous Linux), aller dans le répertoire de Symfony puis lancer **php app/console generate :bundle**

Convention du namespace : NomDuGroupeDeProjet/NomDuProjet**Bundle**

Aux questions posées, on appliquera les options par défaut en cliquant sur *entrée*. On choisira **annotation** pour les formats de configurations et on générera la structure complète en entrant **yes** après les formats.

J'ai choisi dans cette présentation, annotation pour le format de routage. Il en existe trois autres. Voici brièvement les différentes solutions suivi du lien vers la documentation officielle.

- Le format annotation permet d'ajouter une route directement au-dessus des méthodes réalisées. Ce format se rapproche des annotations de PHP ou Java. Assez souple, il faut toutefois se rappeler des routes réalisées dans les différents contrôleurs.
- Le format de configuration yaml centralise les routes dans un fichier de routage. Il faudra faire attention aux espaces si vous utilisez ce format, source fréquente d'erreurs. Notation concise.
- Le format xml est centralisé mais « verbeux ».
- Le format php est lui aussi centralisé et fonctionne par l'ajout de route à une collection. Il nécessite toutefois plus d'écriture que yaml et moins lisible pour comparer les routes.

Conclusion : yaml pour une gestion centralisée, annotation pour l'inverse. Il sera toutefois obligatoire d'utiliser le fichier de routage pour des redirections. Son format sera sous YAML même si vous utilisez des annotations. L'annotation yaml risque d'être toutefois plus performante.

<http://symfony.com/fr/doc/current/book/routing.html>

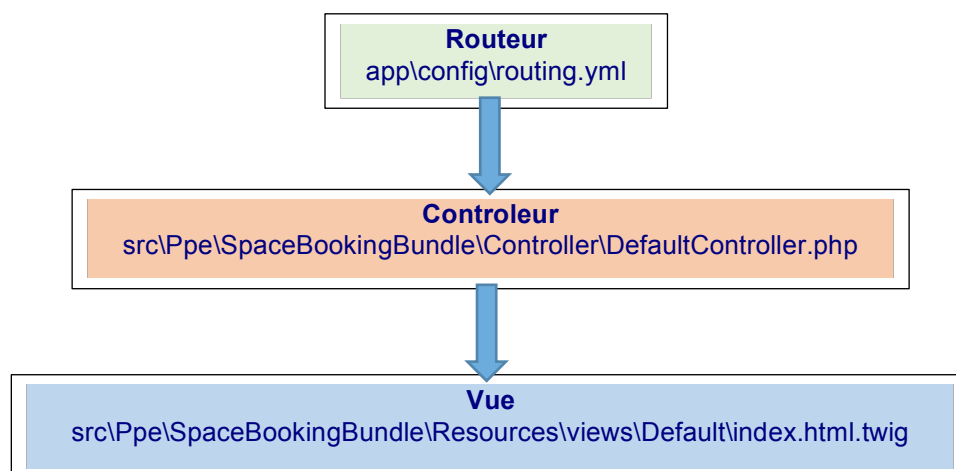
Fonctionnement

Premier exemple

Dans un souci de simplification, toutes les étapes ne sont pas représentées, notamment celle du moteur de Symfony.

Avec un bundle PpeSpaceBooking, on tape l'URL. Le routeur détermine le contrôleur en charge de la route. Le contrôleur exécute la méthode qui gère l'URL /hello/test en renvoyant les paramètres nécessaires à la vue.

http://localhost/symfony/web/app_dev.php/hello/test



Dans le routeur sous **app/config/routing.yml**, le code généré est le suivant :

```
ppe_space_booking:
  resource: "@PpeSpaceBookingBundle/Controller/"
  type:     annotation
  prefix:   /
```

Par défaut, pour les URL de type http://localhost/symfony/web/app_dev.php, on appelle le contrôleur de PpeSpaceBooking. Le contrôleur fonctionne avec des annotations.

Dans le contrôleur **src/ ppe/spaceBookingBundle /controleur/DefaultControleur.php**

```
<?php

namespace Ppe\SpaceBookingBundle\Controller;

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

class DefaultController extends Controller
{
    /**
     * @Route("/hello/{name}")
     * @Template()
     */
    public function indexAction($name)
    {
        return array('name' => $name);
    }
}
```

Les annotations, sous forme d'@ dans les commentaires permettent d'indiquer au contrôleur ce qu'il doit faire.

La route correspond à l'URL entrée. Le template, ici sans paramètre indique qu'il devra charger la vue correspondant au nom de la fonction, soit **index.html.twig**. Le contrôleur envoie un tableau à la vue contenant le name de l'URL sous la forme d'une entrée d'un tableau associatif. La valeur de la clé est quasi la seule information partagée entre le contrôleur et la vue associée (couplage faible).

Dans la vue

```
Hello {{ name }}!
```

Une balise Twig qui permet d'afficher le paramètre transmit par le contrôleur (une de ses méthodes d'action).

Si on ajoute les balises HTML, on aura une barre de Symfony permettant d'obtenir des informations supplémentaires.

localhost/symfony/web/app_dev.php/voyage/Jennifer

Bienvenue sur Travel Advisor Jennifer

Status	200 OK
Controller	DefaultController ::indexAction
Route name	td_traveladvisor_default_index_1
Has session	no

2.6.3 php 47d7b9 200 1 319 ms 4.2 MB 0 0 0

Autre exemple

En tapant l'URL **/waza/wiwi**

Le contrôleur (DefaultController.php)

```
/**
 * @Route("/waza/{name}")
 * @Template()
 */
public function meumeuAction($name)
{
    return array('name' => $name);
}
```

Il est possible de surcharger le nom des routes par défaut. La route précédente portera le nom `ppe_spacebooking_default_meumeu` (documentation : une route définie avec l'annotation `@Route` reçoit un nom prédéfini, composé du nom du bundle, du nom du contrôleur et du nom de l'action). Pour attribuer un nom, il suffit d'ajouter l'attribut `name`.

```
/**
 * @Route("/waza/{name}", name="myMeumeu")
 * @Template()
 */
public function meumeuAction($name)
{
    return array('name' => $name);
}
```

La vue (meumeu.html.twig)

```
<html>
<body>
    {{ name }} est une vache
</body>
</html>
```

On obtient l'affichage **wiwi est une vache**.

Le contrôleur

On peut définir plusieurs contrôleurs

Un contrôleur `MonController` qui prend les URL de la forme `/test1/xx`

```
<?php

namespace Ppe\SpaceBookingBundle\Controller;

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

class MonController extends Controller
{
    /**
     * @Route("/test1/{name}")
     * @Template()
     */
    public function indexAction($name)
```

```
{
    return array('name' => $name);
}
```

Symfony va rechercher dans le contrôleur par défaut les routes, s'il ne trouve rien, il passe aux autres contrôleurs. La route détermine en partie la méthode à exécuter.

Ce contrôleur va appeler `PpeSpaceBookingBundle\Resources\views\Mon\index.html.twig`

`{name}` est récupéré de l'URL et devient un paramètre. Par défaut, si un paramètre est spécifié, il est obligatoire.

Il est possible de mettre plusieurs paramètres à la suite séparés par le caractère de votre choix.

Ce contrôleur affiche une vue, vous verrez certainement dans d'autres exemples (par défaut, la méthode `render` est inutile avec l'annotation `@Template`).

Redirection

La redirection permet d'effectuer des actions puis de rediriger vers une autre méthode.

```
public function indexAction($maValeur)
{
    return $this->redirect($this->generateUrl('homepage', array('maVar' => 'maValeur')));
}
```

La fonction va rechercher le nom 'homepage' dans `routing.yml`

```
ppe_space_booking:
    resource: "@PpeSpaceBookingBundle/Controller/"
    type:     annotation
    prefix:   /
    homepage:
        path: /meumeu/{maVar}
```

puis générer l'URL `/meumeu/maValeur` et rechercher un contrôleur correspondant.

Forward ou transfert

La méthode `forward` permet d'exécuter une sous requête. Symfony appelle la méthode correspondante et renvoie la réponse.

```
public function indexAction($name)
{
    $response = $this->forward(PpeSpaceBookingBundle:Mon:index', array(
        'name' => $name,
        'color' => 'green',
    ));
    return $response;
}
```

`Mon` correspond au nom du contrôleur

`Index` correspond au nom de la méthode

Si vous avez des problèmes avec vos routes, vous pouvez utiliser dans la console `app/console router:debug`. Vous obtiendrez une liste de vos routes avec les URL correspondantes.

td_test_default_index	ANY	ANY	ANY	/hello/{name}
td_traveladvisor_admin_index	ANY	ANY	ANY	/admin
td_traveladvisor_admin_redirection	ANY	ANY	ANY	/admin/redirection
td_traveladvisor_default_ajoutetablissementatequipement	ANY	ANY	ANY	/equipement/ajoutEtablissement/{idEtablissement}
td_traveladvisor_default_ajoutequipementatetablissement	ANY	ANY	ANY	/etablissement/ajoutEquipement/{libelle}
td_traveladvisor_default_afficheravis	ANY	ANY	ANY	/avis
td_traveladvisor_default_ajoutetablissement	ANY	ANY	ANY	/ajoutEtablissement
td_traveladvisor_default_ajoutavis	ANY	ANY	ANY	/ajoutAvis
td_traveladvisor_default_index	ANY	ANY	ANY	/voyage
td_traveladvisor_default_prenom	ANY	ANY	ANY	/voyage/{name}
td_traveladvisor_default_avis	ANY	ANY	ANY	/avis/{nombre}
td_traveladvisor_default_ajouter	ANY	ANY	ANY	/ajoutMembre
td_traveladvisor_default_afficher	ANY	ANY	ANY	/afficherMembre
homepage	ANY	ANY	ANY	/app/example
zoneVisiteur	ANY	ANY	ANY	/voyage/visiteur

Les annotations

Les annotations simplifient principalement le routage dans Symfony. Les annotations sont présentes dans de nombreux de langages.

Toute annotation utilisée dans le contrôleur doit être importée sous la forme :

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
```

Il existe plusieurs annotations :

- @Route
- @Method
- @Template
- @ParamConverter
- @Cache

@Route

Il est possible de définir plusieurs URL pour une route

```
/**
 * @Route("/test1/")
 * @Route("/test2/")
 */
public function indexAction($name)
```

On peut récupérer des variables comme vu précédemment

On peut ajouter des attributs

```
/**
 * @Route("/test/{numPage}", name="monTest", defaults={"numPage"= 1}, , requirements={"numPage"
 = "\d+ "})
 */
public function indexAction($numPage)
```

requirements utilise une expression régulière pour le paramètre id.

defaults permet de préciser un format ou une valeur par défaut d'un attribut, il ne devient plus obligatoire

name correspond au nom de la route, utile pour déboguer.

@Method

```
/**
 * @Route("/test1/")
 * @Method({"GET", "POST"})
 */
public function indexAction($name)
```

La route n'est utilisée que si elle provient d'une requête http GET ou POST.

@method n'est utilisable qu'avec @route

@Template

Il permet de rediriger vers une vue le contrôleur différent de celui par défaut.

```
class DefaultController extends Controller
{
 /**
  * @Route("/hello/{name}", name="maRedirection")
  * @Template("PpeSpaceBookingBundle:Mon:meumeu.html.twig")
  */
 public function indexAction($name)
```

Par défaut, Ppe\SpaceBookingBundle\Resources\views\Default\index.html.twig

Mais ici, Ppe\SpaceBookingBundle\Resources\views\Mon\meumeu.html.twig

@ParamConverter

Convertit des paramètres de requêtes en objet.

```
/**
 * @Route("/monUrl/{id}")
 * @ParamConverter("post", class=" PpeSpaceBookingBundle:Post")
 */
public function showAction(Post $post){
}
```

Le @ParamConverter crée un objet. On utilisera cette balise pour récupérer les données de la base.

@Cache

Définit les options de cache des pages.

```
/**
 * @Cache(expires="tomorrow")
 */
public function indexAction(){
}
```

On peut utiliser cette annotation quand on récupère des données conséquentes de la base pour éviter de surcharger le serveur.

Autre exemple

```
@Cache(expires="+2 days")
```

```
@Cache(maxage="15")
```

Autorise le serveur à renvoyer la même page pendant le temps défini.

```
@Cache(smaxage="15")
```

Idem mais pour les caches partagés.

Twig pour les vues

Les variables

On a vu précédemment comment afficher une variable

```
{{maVariable}}
```

Même principe pour un objet avec utilisation implicite des accesseurs et modificateurs

```
{{ monObjet.maVariable }}
```

Les conditions

```
{% if maVariable > 24 %}  
    Il y a plus de 24 personnes  
{% elseif maVariable < 24 %}  
    Il y a moins de 24 personnes  
{% else %}  
    Il y a 24 personnes  
{% endif %}
```

Équivalent du isset en PHP

```
{% if monObjet.maVariable is defined %}
```

Teste si la variable contient une valeur

```
{% if monObjet.maVariable is empty %}
```

Les boucles

```
{% for i in 1..10 %}  
    indice {{ i }}  
{% endfor %}
```

```
{% for unObjet in maCollection %}  
    {{ unObjet.uneVariable }}  
{% endfor %}
```

Les filtres

<http://twig.sensiolabs.org/doc/filters/index.html>

On peut utiliser des filtres pour modifier l'affichage ou récupérer la valeur d'une variable.

```
{{ 'welcome'|upper }}  
  
{% if users|length > 10 %}  
    ...  
{% endif %}  
  
{% filter lower %}  
{{ma Variable}}  
Mon texte  
Mon second TEXTE  
{% endfilter %}
```

Layout

Il existe un mécanisme qui permet de réaliser des héritages de template. Il suffit de définir un template « layout » puis d'hériter celui-ci. Par défaut le fichier `app/Resources/views/base.html.twig` n'est pas équivalent à celui-ci.

```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>{% block title %}titre générique{% endblock %}</title>
  </head>
  <body>

    Divers contenus

    <div id="content">
      {% block lesSalles %}{% endblock %}
    </div>
  </body>
</html>
```

Le layout enfant

```
{# src/Ppe/Spacebooking/Resources/views/Default/Spacebooking/maPage.html.twig #}
{% extends '::base.html.twig' %}

{% block title %}Mes salles{% endblock %}

{% block lesSalles %}
  {% for salle in salle_entries %}
    <h2>{{ salle.titre }}</h2>
    <p>{{ salle.description }}</p>
  {% endfor %}
{% endblock %}
```

On réalise un template qui va redéfinir certains blocs du template de base. Symfony évaluera quel bloc est à remplacer.

La sortie

```
<!DOCTYPE html>
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Mes salles</title>
  </head>
  <body>

    Divers contenus

    <div id="content">
      <h2>Mandelbrot</h2>
      <p>une salle studieuse</p>

      <h2>Vico le roi</h2>
      <p>une salle gourmande !</p>
    </div>
  </body>
</html>
```

Generation d'url

```
<a href = « {{path('homepage')}} »> page d'accueil </a>
```

Appelle la route homepage de routing.yml.

On peut ajouter un ou plusieurs paramètres.

```
<a href = « {{path('homepage', {'maVar' : 'var'})}} »> page d'accueil </a>
```

Doctrine pour le modèle

Doctrine utilise le mapping avec des objets PHP contenant des annotations.

Génération d'une classe

On utilise la console (PowerShell sous Windows par exemple) pour générer les fichiers nécessaires. À la racine de l'application en mode console :

```
www\symfony> php app/console generate:doctrine:entity
```

```
Client(id, nom, prenom, reference)
```

```
The Entity shortcut name: PpeSpaceBookingBundle:Client
```

```
Determine the format to use for the mapping information.
```

```
Configuration format (yml, xml, php, or annotation) [annotation]: annotation
```

```
Instead of starting with a blank entity, you can add some fields now.  
Note that the primary key will be added automatically (named id).
```

```
Available types: array, simple_array, json_array, object,  
boolean, integer, smallint, bigint, string, text, datetime, datetimetz,  
date, time, decimal, float, blob, guid.
```

```
New field name (press <return> to stop adding fields): nom
```

```
Field type [string]: string
```

```
Field length [255]: 30
```

```
New field name (press <return> to stop adding fields): prenom
```

```
Field type [string]: string
```

```
Field length [255]: 30
```

```
New field name (press <return> to stop adding fields): reference
```

```
Field type [string]: integer
```

```
Do you want to generate an empty repository class [no]? yes
```

```
Summary before generation
```

```
You are going to generate a "PpeSpaceBookingBundle:Client" Doctrine2 entity  
using the "annotation" format.
```

```
Do you confirm generation [yes]? yes
```

```
Entity generation
```

```
Generating the entity code: OK
```

```
You can now start using the generated code!
```

Résultat dans `www\symfony\src\Ppe\SpaceBookingBundle\Entity`.

On remarque de nouvelles annotations `@ORM` qui indiquent à Doctrine ce qu'il doit faire.

```
<?php  
  
namespace Ppe\SpaceBookingBundle\Entity;  
  
use Doctrine\ORM\Mapping as ORM;  
  
/**  
 * Client  
 *  
 * @ORM\Table() */
```

```

* @ORM\Entity(repositoryClass="Ppe\SpaceBookingBundle\Entity\ClientRepository")
*/
class Client
{
    /**
     * @var integer
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string
     *
     * @ORM\Column(name="nom", type="string", length=30)
     */
    private $nom;

    /**
     * @var string
     *
     * @ORM\Column(name="prenom", type="string", length=30)
     */
    private $prenom;

    /**
     * @var integer
     *
     * @ORM\Column(name="reference", type="integer")
     */
    private $reference;

    /**
     * Get id
     *
     * @return integer
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * Set nom
     *
     * @param string $nom
     * @return Client
     */
    public function setNom($nom)
    {
        $this->nom = $nom;

        return $this;
    }

    /**
     * Get nom
     *
     * @return string
     */
    public function getNom()
    {
        return $this->nom;
    }

    /**
     * Set prenom
     *
     * @param string $prenom
     * @return Client
     */
    public function setPrenom($prenom)
    {
        $this->prenom = $prenom;

        return $this;
    }

    /**
     * Get prenom
     *
     * @return string
     */
    public function getPrenom()
    {
        return $this->prenom;
    }

    /**
     * Set reference

```

```

*
* @param integer $reference
* @return Client
*/
public function setReference($reference)
{
    $this->reference = $reference;

    return $this;
}

/**
* Get reference
*
* @return integer
*/
public function getReference ()
{
    return $this->reference;
}
}

```

Ce fichier est modifiable.

Les annotations

docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html

@ORM\Entity(repositoryClass="Ppe\SpaceBookingBundle\Entity\ClientRepository")

Définit la classe comme une entité à enregistrer par Doctrine, repositoryClass indique la classe (et donc le fichier) de mapping utilisé pour cette classe Entity.

@ORM\Table(name= « maTable»)

Définit le nom de la table dans la base de données - optionnelle

@ORM\Column(name="prenom", type="string", length=30)

L'exemple parle de lui-même.

Il est possible de rajouter plusieurs paramètres :

unique=true, nullable=true, precision=5, scale=2

Génération des tables

Il faut indiquer les paramètres du SGBDR dans `www\symfony\app\config\parameters.yml` en indiquant le nom de la base de données utilisée.

La database peut être créée ultérieurement.

Création de la BD

```

www\symfony> php app/console doctrine:database:create
Created database for connection named `ppeSpaceBooking`

```

Affichage des requêtes à exécuter

```

www\symfony> php app/console doctrine:schema:update --dump-sql
CREATE TABLE Client (id INT AUTO_INCREMENT NOT NULL, nom VARCHAR(30)
NOT NULL, prenom VARCHAR(30) NOT NULL, reference INT NOT
NULL, PRIMARY KEY(id)) DEFAULT CHARACTER SET utf8 COLLATE
utf8_unicode_ci ENGINE = InnoDB;

```

Création des tables

```

www\symfony> php app/console doctrine:schema:update --force
Updating database schema...
Database schema updated successfully! "1" queries were executed

```

Enregistrement d'un objet dans le SGBDR

Dans un contrôleur

```
<?php

//en plus des autres importations
use Ppe\SpaceBookingBundle\Entity\Client;

/**
 * @Route("/enregistrement")
 */
public function ajouterAction() {

    // Création de l'entité
    $client = new Client();
    $client->setNom('Dumont');
    $client->setPrenom('Paul');
    $client->setReference(130303030);

    //Récupération de l'EntityManager
    $em = $this->getDoctrine()->getManager();

    //gestion de $client par l'ORM
    $em->persist($client);

    //l'ORM regarde les objets qu'il gère pour savoir s'ils doivent être persistés
    $em->flush();

    //redirection ou affichage d'une vue
}
```

Il suffit donc de récupérer l'EntityManager, de le persister et de l'insérer avec flush.

Pour optimiser les performances, il est possible de persister un ensemble d'objets puis de réaliser un flush qui va enregistrer globalement les changements.

On pourra cliquer sur l'onglet tout à droite de Symfony pour voir les requêtes exécutées.

Récupérer un objet

```
$id = 1 ;
$monClient = em->find('PpeSpaceBookingBundle :Client', $id);
```

On récupère le client qui possède la clef primaire 1.

Méthodes de l'EntityManager

Plusieurs méthodes permettent de manipuler les objets gérés par l'ORM. On trouvera la liste complète ci-dessous.

doctrine-project.org/api/orm/2.1/class-Doctrine\ORM\EntityManager.html

Pour un objet \$a

```
//annule les persist
```

```
$em->clear() ;  
//même chose sur un objet unique  
$em->detach($a) ;  
//supprime l'objet de la BD  
$em->remove($a) ;  
//MAJ de l'objet en fonction de la BD  
$em->refresh($a) ;
```