Symfony

Tout le cours s’articule autour de la création d’une plateforme d’échange.

Table des matières

[Installer symfony 5](#_Toc468292236)

[Découvrir Symfony 5](#_Toc468292237)

[Les répertoires : 5](#_Toc468292238)

[Le contrôle frontal : 5](#_Toc468292239)

[Deux environnements de travail : 5](#_Toc468292240)

[Que contrôle le contrôle frontal ? 6](#_Toc468292241)

[L’architecture conceptuelle : 6](#_Toc468292242)

[Les Bundles : 7](#_Toc468292243)

[Structure d’un Bundle : 8](#_Toc468292244)

[Utiliser la console pour créer un Bundle : 8](#_Toc468292245)

[Créer le Bundle : 8](#_Toc468292246)

[Symfony enregistre le Bundle près du Kernel : 8](#_Toc468292247)

[Symfony enregistre nos routes auprès du Routeur : 9](#_Toc468292248)

[Premier script 10](#_Toc468292249)

[Créer le route : 10](#_Toc468292250)

[Créer le contrôleur : 11](#_Toc468292251)

[Les templates (ou vues) Twig : 12](#_Toc468292252)

[Appeler le template depuis le controller : 13](#_Toc468292253)

[Notre objectif : créer une plateforme d’annonce : 14](#_Toc468292254)

[Un peu de nettoyage : 14](#_Toc468292255)

[Schéma de développement sous Symfony : 15](#_Toc468292256)

[Le routeur de Symfony 16](#_Toc468292257)

[Le fonctionnement : 16](#_Toc468292258)

[Fonctionnement du routeur : 16](#_Toc468292259)

[Les routes de base : 17](#_Toc468292260)

[Créer une route avec des paramètres : 18](#_Toc468292261)

[Les routes avancées : 19](#_Toc468292262)

[Paramètres facultatifs : 19](#_Toc468292263)

[Utiliser des paramètres systèmes : 20](#_Toc468292264)

[Ajouter un préfixe lors de l’import de nos routes : 20](#_Toc468292265)

[Générer des URL : 20](#_Toc468292266)

[Pourquoi générer des URL ? 20](#_Toc468292267)

[Comment générer des URL : 21](#_Toc468292268)

[Application : les routes de notre plateforme : 22](#_Toc468292269)

[Les contrôleurs avec Symfony 23](#_Toc468292270)

[Le rôle du contrôleur : 23](#_Toc468292271)

[L’objet Request : 24](#_Toc468292272)

[Les paramètres contenus dans les routes : 24](#_Toc468292273)

[Les paramètres hors routes : 24](#_Toc468292274)

[Les autres méthodes de l’objet Request : 26](#_Toc468292275)

[Réponses et vues : 27](#_Toc468292276)

[Réponse et redirection : 28](#_Toc468292277)

[Changer le Content-type de la réponse : 29](#_Toc468292278)

[Manipuler la session : 30](#_Toc468292279)

[Le contrôleur de note plateforme : 32](#_Toc468292280)

[Le moteur de template Twig 34](#_Toc468292281)

[Les templates Twig : 34](#_Toc468292282)

[Intérêt : 34](#_Toc468292283)

[Page web, email … 34](#_Toc468292284)

[En pratique : 34](#_Toc468292285)

[Afficher des variables : 35](#_Toc468292286)

[Twig et la sécurité : 36](#_Toc468292287)

[Les variables globales : 37](#_Toc468292288)

[Structure et contrôle des expressions : 38](#_Toc468292289)

[Les structures de contrôle : 38](#_Toc468292290)

[Hériter et inclure des templates : 40](#_Toc468292291)

[La pratique : 40](#_Toc468292292)

[Le nom du template père : 41](#_Toc468292293)

[Organiser son code en triple héritage : 41](#_Toc468292294)

[Héritage ou inclusion ? 42](#_Toc468292295)

[La pratique : 43](#_Toc468292296)

[Inclusion de contrôleur : 43](#_Toc468292297)

[Application : 44](#_Toc468292298)

[Layout général : 45](#_Toc468292299)

[Layout du Bundle : 46](#_Toc468292300)

[Templates finaux : 46](#_Toc468292301)

[Installer un Bundle avec Composer 47](#_Toc468292302)

[C’est quoi ? 47](#_Toc468292303)

[L’installation : 47](#_Toc468292304)

[Installer un Bundle : 48](#_Toc468292305)

[Installer un Bundle : 50](#_Toc468292306)

[Gérer l’autoload : 50](#_Toc468292307)

[Services, théories et créations 51](#_Toc468292308)

# Installer symfony

On doit télécharger un certificat de sécurité : présent dans le dossier symfony.

Dans php.ini on doit spécifier dans CURL =, le chemin d’accès absolu vers ce fichier. Ensuite on peut enfin créer un fichier de travail avec « php symfony.phar new test ».

Sous Linux, une partie sur les droits d’accès est à revoir sur le cours.

# Découvrir Symfony

## Les répertoires :

* App : contient tout ce qui concerne le site sauf son code source.
* Bin : contient les exécutables dont nous avons besoin, comme des commandes PHP.
* Tests : contient tous les tests de l’application, pas évoqué ici mais utile.
* Var : tout ce qui est utile au bon fonctionnement. On n’écrit pas dedans nous-même.
* Vendor : contient toutes les bibliothèques externes au projet.
* Web : contient tout ce qui est destiné au visiteur : image, css, JS … C’est aussi le seul qui sera accessible par celui-ci.

On utilisera donc toujours Symfony/web, pour être sûr que le visiteur n’accède qu’à celui-ci. On passe donc du temps dans le répertoire **SRC** pour travailler sur les **BUNDLES**. Au répertoire **APP** pour configurer les applications. Et **VENDOR** pour installer des bundles téléchargés.

### Le contrôle frontal :

C’est l’index.php, la page d’entrée du site. Avec Symfony on en a deux :

* Un pour les visiteurs : app.
* Un pour le dév’ app\_dev.

### Deux environnements de travail :

Chaque personne n’a pas besoin des mêmes informations sur la page :

* Un développeur a besoin d'informations sur la page afin de l'aider à développer. En cas d'erreur, il veut tous les détails pour pouvoir déboguer facilement. Il n'a pas besoin de rapidité.
* Un visiteur normal n'a pas besoin d'informations particulières sur la page. En cas d'erreur, l'origine de celle-ci ne l'intéresse pas du tout, il veut juste retourner d'où il vient. Par contre, il veut que le site soit le plus rapide possible à charger.

Si on essaye d’accéder à la page app.php/\_profiler au lieu de app\_dev, on n’a rien, parce qu’on n’a pas défini pour l’url \_profiler. On obtient une error 404. Quand on essaye d’aller sur une page qui n’existe pas on obtient une tout autre page.

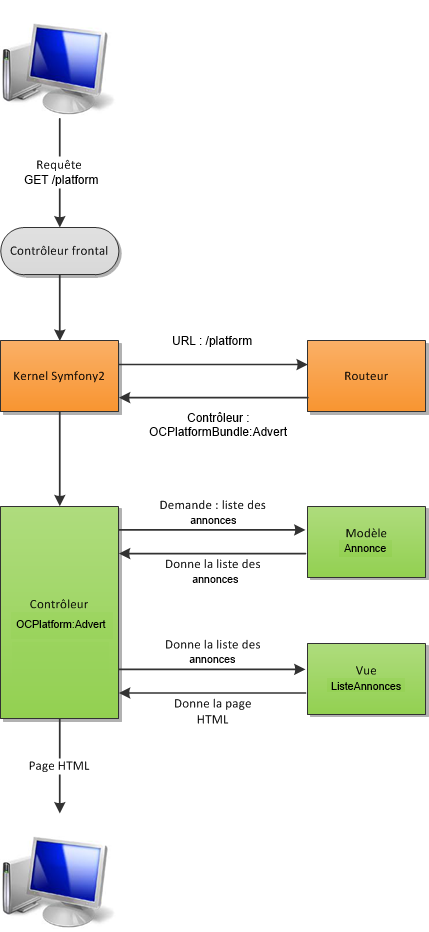
Pour trouver les erreurs on regarde le fichier var/logs/prod.log.

### Que contrôle le contrôle frontal ?

Ici on se limite à appeler le noyau (kernel) de Symfony.

## L’architecture conceptuelle :

Le modèle MVC, on peut voir le chemin d’une requête :



* **Le contrôleur** : son rôle est de générer la réponse à la requête http demandée. Il analyse et traite la réponse de l’utilisateur.
* **Le modèle** : gère les données et le contenu.
* **La vue** : affiche les pages.

L’utilisateur fait une requête 🡺 le contrôleur les reçoit et demande au modèle 🡺 il les renvoie au contrôleur 🡺 qui envoie à la vue et l’affiche 🡺 le contrôleur a son information.

## Les Bundles :

On regroupe les fonctionnalités par Bundle, chaque Bundle gère une tâche bien précise.

Exemple : le Bundle blog contiendra contrôleur, modèle, vue, code JS et CSS qui concerne uniquement le blog.

* Un bundle Utilisateur, qui va gérer les utilisateurs ainsi que les groupes, intégrer des pages d'administration de ces utilisateurs, et des pages classiques comme le formulaire d'inscription, de récupération de mot de passe, etc.
* Un bundle Blog, qui va fournir une interface pour gérer un blog sur le site. Ce bundle peut utiliser le bundle Utilisateur pour faire un lien vers les profils des auteurs des articles et des commentaires.
* Un bundle Boutique, qui va fournir des outils pour gérer des produits et des commandes dans un site e-commerce par exemple.

On peut donc importer des exporter des fonctionnalités, des Bundles que l’on réutilisera par la suite.

Quand on fait son programme, pour être rapide si on ne souhaite pas le partager on fait souvent dans un seul et même Bundle appelé souvent Core ou App.  
Les Bundles de la communauté sur ce site : [les Bundles](http://knpbundles.com).

**Quelques Bundles utiles :**

* [FOSUserBundle](http://knpbundles.com/FriendsOfSymfony/FOSUserBundle) pour gérer ses utilisateurs.
* [FOSCommentBundle](http://knpbundles.com/FriendsOfSymfony/FOSCommentBundle) gère les commentaires.
* [GravatarBundle](http://knpbundles.com/ornicar/GravatarBundle) gère les avatars des utilisateurs depuis Gravatar.

### Structure d’un Bundle :

/Controller          | Contient vos contrôleurs

/DependencyInjection | Contient des informations sur votre bundle (chargement automatique de la configuration par exemple)

/Entity              | Contient vos modèles

/Form                | Contient vos éventuels formulaires

/Resources

-- /config             | Contient les fichiers de configuration de votre bundle (nous placerons les routes ici, par exemple)

-- /public             | Contient les fichiers publics de votre bundle : fichiers CSS et JavaScript, images, etc.

-- /views              | Contient les vues de notre bundle, les templates Twig

# Utiliser la console pour créer un Bundle :

Symfony utilise des commandes disponibles sur la console et non le navigateur.

Pour faire une commande, on se place dans le dossier de Symfony et on exécute une commande PHP : php bin/console.

Grace à ça on pourra par exemple, créer une BDD ou en vider une sans phpmyadmin, vider le cache etc …

Ici on va générer le code de base pour les Bundles qui est identiques à chacun. Le code bin/console fait la même chose que app.php mais définit la requête comme venant de la console.

## Créer le Bundle :

On utilise toujours la même syntaxe :

Php bin/console generate :bundle

Pour nommer son Bundle comme ainsi : OC/PlatformBundle, on a OC qui est la présentation de site, entreprise … en racine, puis le nom du Bundle puis « Bundle » qui est obligatoire. Le nom final est sans slash : OCPlatformBundle.

Ensuite on a le choix du format (plus tard) ici yml.

### Symfony enregistre le Bundle près du Kernel :

La configuration se fait dans App, ainsi on a app/AppKernel.php dans lequel il faut indiquer que l’on veut charger notre Bundle.

Les premières lignes indiquent les Bundles à charger pour la production et les suivants (dans le if) pour le développement. Le générateur de Bundle a généré lui-même une ligne supplémentaire en production avec notre nouveau Bundle.

### Symfony enregistre nos routes auprès du Routeur :

Routeur : défini quel contrôleur appliqué en fonction de l’URL appelé. Pour cela il utilise les routes.

Chaque Bundle a ses propres routes, que l’on peut voir ici : src/OC/PlatformBundle/Resources/config/routing.yml.

Pour le moment il n’y en a que une :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_homepage:

path: /

defaults: { \_controller: OCPlatformBundle:Default:index }

Les routes ne se chargent pas toute seul, il faut dire au routeur de les charger. Ca se fait dans « application »

# app/config/routing.yml

oc\_platform:

resource: "@OCPlatformBundle/Resources/config/routing.yml"

prefix: /

# ...

**Ce qu’il faut retenir :**

Ce qu'il faut retenir de tout cela, c'est que pour qu'un bundle soit opérationnel, il faut :

* Son code source, situé dans src/Application/Bundle, et dont le seul fichier obligatoire est la classe à la racine OCPlatformBundle.php ;
* Enregistrer le bundle dans le noyau pour qu'il soit chargé, en modifiant le fichier app/AppKernel.php ;
* Enregistrer les routes (si le bundle en contient) dans le Routeur pour qu'elles soient chargées, en modifiant le fichier app/config/routing.yml.

Ces tâches se font automatiquement avec le générateur. Mais elles sont à faire manuellement si on ne l’utilise pas.

# Premier script

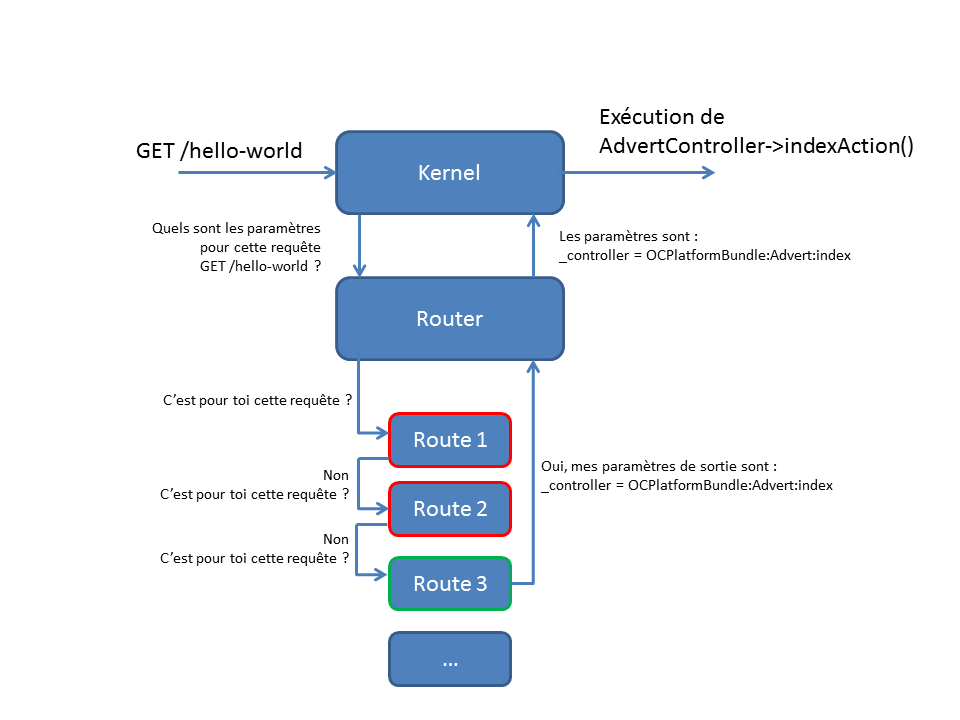
## Créer le route :

On va dans src/oc/PlateformBundle.

Pour créer une page il faut définir l’URL à laquelle elle est disponible. Il faut donc créer le route de cette page.

**L’objectif du routeur est de faire correspondance entre l’URL et des paramètres.**

Exemple : on pourrait avoir une route qui dit : quand l’URL est /hello-world, le contrôleur à exécuter est « Advert ».



Donc dans notre Bundle OC/PlateformBundle on exécute le contrôleur Advert.

# src/OC/PlatformBundle/Resources/config/routing.yml

hello\_the\_world:

path: /hello-world

defaults: { \_controller: OCPlatformBundle:Advert:index }

Attention l'indentation se fait avec 4 espaces par niveau, et non avec des tabulations ! Je le précise parce qu'un jour vous ferez l'erreur (l'inévitable ne peut être évité), et vous me remercierez de vous avoir mis sur la voie. Et cela est valable pour tous vos fichiers YAML (.yml).

Attention également, il semble y avoir des erreurs lors des copier-coller depuis le tutoriel vers les fichiers .yml. Si vous rencontrez une obscure erreur, pensez à bien définir l'encodage du fichier en « UTF-8 sans BOM » et à supprimer les éventuels caractères non désirés. C'est un bug étrange qui provient du site, mais dont on ne connaît pas l'origine. L'esquive est de toujours recopier les exemples YAML que je vous donne, et de ne pas les copier-coller.

* Hello\_the\_world est le nom de la route. Il doit être **unique**.
* Path est l’URL à laquelle on souhaite qu’il soit accessible.
* Defaults correspond aux paramètres de la route.
  + \_controller correspond à l’action que l’on veut exécuter et au contrôleur que l’on va appeler. Action = index, contrôleur = Advert.

On donne la valeur OCPlateformBundle:Advert:index à \_controller.

* OCPlatformBundle est le nom du Bundle, celui dans lequel Symfony ira chercher le contrôleur.
* Advert est le nom du contrôleur à ouvrir : src/OC/PlatformBundle/Controller/AdvertController.php.
* Index est le nom de la méthode à exécuter au sein du contrôleur.

On a déjà vu dans app/config/routing.yml que le fichier route du Bundle est déjà inclus dans la configuration générale.

On aurait pu ajouter hello\_the\_world directement dans le routing.yml mais on irait à l’encontre des principes des Bundles.

Au moins plus tard on pourra modifier le fichier : **src/OC/PlatformBundle/Resources/config/routing.yml** au lieu de **app/config/routing.yml**.

## Créer le contrôleur :

Dans le Bundle, les contrôleurs se trouvent dans « Controller ». On a comme convention d’avoir le suffixe « controller » dans les fichiers :

Src/OC/PlatformBudle/Controller/AdvertController.php.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController

{

public function indexAction()

{

return new *Response*("Notre propre Hello World !");

}

}

* Ligne 5 : on se place dans le namespace des contrôleurs de notre bundle. Suivez juste la structure des répertoires dans lequel se trouve le contrôleur.
* Ligne 7 : notre contrôleur va utiliser l'objet Response, il faut donc le définir grâce au use.
* Ligne 9 : le nom de notre contrôleur respecte le nom du fichier pour que l'*autoload* fonctionne.
* Ligne 11 : on définit la méthode indexAction(). N'oubliez pas de mettre le suffixe Action derrière le nom de la méthode.
* Ligne 13 : on crée une réponse toute simple. L'argument de l'objet Response est le contenu de la page que vous envoyez au visiteur, ici « Notre propre Hello World ! ». Puis on retourne cet objet.

Si on parle de l’action « index », le contrôleur doit définir la méthode indexAction().

## Les templates (ou vues) Twig :

**Twig est un moteur de template** : un script qui affiche le contenu dynamiquement de note page HTML. Chaque moteur de template a son propre langage.

Exemple : code 1 en PHP, code 2 en langage Twig

<!DOCTYPE html>

<html>

<head>

<title>Bienvenue dans Symfony2 !</title>

</head>

<body>

<h1><?php echo $titre\_page; ?></h1>

<ul id="navigation">

<?php foreach ($navigation as $item) { ?>

<li>

<a href="<?php echo $item->getHref(); ?>"><?php echo $item->getTitre(); ?></a>

</li>

<?php } ?>

</ul>

</body>

</html>

<!DOCTYPE html>

<html>

<head>

<title>Bienvenue dans Symfony2 !</title>

</head>

<body>

<h1>{{ titre\_page }}</h1>

<ul id="navigation">

{% for item in navigation %}

<li><a href="{{ item.href }}">{{ item.titre }}</a></li>

{% endfor %}

</ul>

</body>

</html>

On va donc utiliser le moteur Twig au lieu d’afficher notre texte de manière standard.

Le répertoire des templates est dans Resources/views.

On va donc y insérer :

{# src/OC/PlatformBundle/Resources/views/Advert/index.html.twig #}

<!DOCTYPE html>

<html>

<head>

<title>Bienvenue sur ma première page avec OpenClassrooms !</title>

</head>

<body>

<h1>Hello World !</h1>

<p>

Le Hello World est un grand classique en programmation.

Il signifie énormément, car cela veut dire que vous avez

réussi à exécuter le programme pour accomplir une tâche simple :

afficher ce hello world !

</p>

</body>

</html>

### Appeler le template depuis le controller :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

// N'oubliez pas ce use :

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

public function indexAction()

{

$content = $this->get('templating')->render('OCPlatformBundle:Advert:index.html.twig');

return new *Response*($content);

}

}

Dans la ligne de $content on a : NomDuBundle :NomDuContrôleur :NomDeLAction.

$this->get(‘templating’) retourne un objet dont le nom ici est templating. La méthode render de l’objet templating permet de récupérer le contenu.

**La toolbar en bas de page apparait quand on a la balise fermante de Body.**

On peut modifier le contenu en ajoutant un tableau :

<?php

$content = $this

->get('templating')

->render('OCPlatformBundle:Advert:index.html.twig', array('nom' => 'winzou'))

;

## Notre objectif : créer une plateforme d’annonce :

La plateforme que nous allons créer est très simple. En voici les grandes lignes :

* Nous aurons des annonces (*advert* en anglais) de mission : développement d'un site internet, création d'une maquette, intégration HTML, etc. ;
* Nous pourrons consulter, créer, modifier et rechercher des annonces ;
* À chaque annonce, nous pourrons lier une image d'illustration ;
* À chaque annonce, nous pourrons lier plusieurs candidatures (*application* en anglais) ;
* Nous aurons plusieurs catégories (Développement, Graphisme, etc.) qui seront liées aux annonces. Nous pourrons créer, modifier et supprimer ces catégories ;
* À chaque annonce, nous pourrons enfin lier des niveaux de compétence requis (Expert en PHP, maîtrise de Photoshop, etc.).
* Au début, nous n'aurons pas de système de gestion des utilisateurs : nous devrons saisir notre nom lorsque nous rédigerons une annonce. Puis, nous rajouterons la couche utilisateur.

Pour éviter les redondances, on supprime :

* Controller/DefaultController.php
* Resources/views/Default
* Oc\_platform\_homepage dans Resources/config/routing.yml

### Un peu de nettoyage :

On va supprimer tous les fichiers « par défaut » de l’application :

* Le contrôleur Controller/DefaultController.php ;
* Le répertoire de vues Resources/views/Default ;
* La route oc\_platform\_homepage dans Resources/config/routing.yml.

Supprimez également tout ce qui concerne le bundle AppBundle,  un bundle de démonstration intégré dans la distribution standard de Symfony2 et dont nous ne nous servirons pas :

* Le répertoire src/AppBundle  ;
* La lignes 19 du fichier app/AppKernel.php, celle qui active le bundle : new AppBundle\AppBundle() ;
* Les lignes 7 à 9 du fichier app/config/routing.yml, celles qui importent le fichier de route du bundle AppBundle (app: resource: "@AppBundle/Controller/" type: annotation" ).

Ainsi on va se concentrer uniquement sur nos Bundles.

### Schéma de développement sous Symfony :

Il faut penser à vider le cache !

* En mode prod’, inutile, on peut vider la cache à la main avec la commande : php bin/console cache :clear –env=prod.
* En mode dév’, Symfony construit une bonne partie du cache à la prochaine page que l’on charge, donc pas besoin de le vider. Au cas où : php bin/console cache :clear.

Parfois ça ne marche qu’au deuxième coup. Sinon pour le vider à la main on supprime les répertoires var/cache/dev et var/cache/prod.

On respecte donc ce schéma :

* Je fais des changements, je teste ;
* Je fais des changements, je teste ;
* Je fais des changements, je teste : ça ne marche pas, je vide le cache : ça marche ;
* Je fais des changements, je teste ;

**En résumé :**

On créée un Bundle, dans lequel on a le modèle MVC ainsi que du code autre. Le bundle est stocké dans **SRC**.

Php bin/console generate :bundle

Dedans on a :

* Controller dans lequel on a le nom du controller et à l’intérieur de celui-ci des méthodes.
* Resources dans lequel on a :
  + Configu pour définir les routes avec le Routeur.
  + Views pour les templates (ici Twig).

On définit les routes dans notre Bundle pour éviter de le faire dans App (qui marcherait aussi). Par contre il faut définir dans route.yml de App que notre Bundle possède des routes :

oc\_platform:

resource: "@OCPlatformBundle/Resources/config/routing.yml"

prefix: /

# Le routeur de Symfony

Comme vu précédemment, le routeur a pour rôle **de définir quel contrôleur appelé en fonction de l’URL**.

## Le fonctionnement :

Le paramètre qui nous intéresse le plus est *\_controller* qui correspond au contrôleur à exécuter.

On va maintenant travailler avec cette feuile de route :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_home:

path: /platform

defaults: { \_controller: OCPlatformBundle:Advert:index }

oc\_platform\_view:

path: /platform/advert/{id}

defaults: { \_controller: OCPlatformBundle:Advert:view }

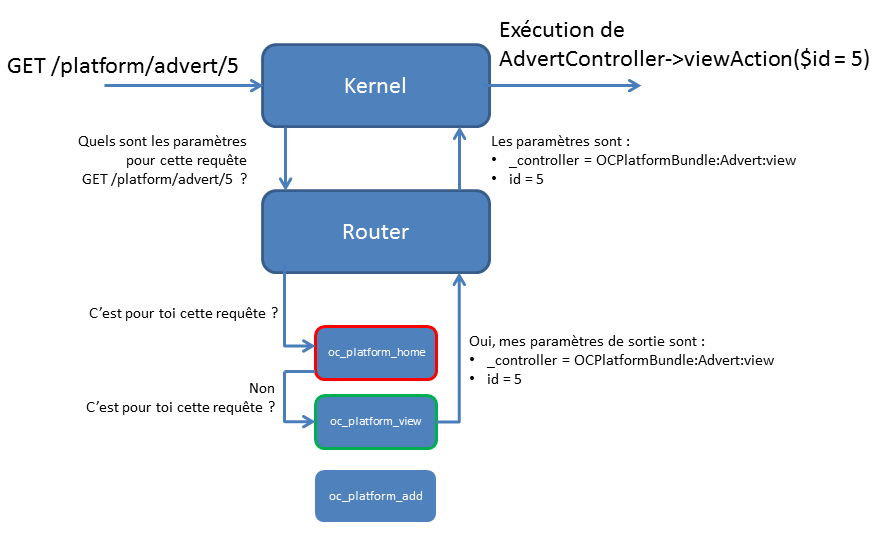
oc\_platform\_add:

path: /platform/add

defaults: { \_controller: OCPlatformBundle:Advert:add }

### Fonctionnement du routeur :

Pour chaque route on a donc une entrée : ‘path’ et une sortie ‘defaults’.



Pour chaque page, il est possible de voir toutes les routes que le routeur essaye une à une et celle qu’il utilise au final. C’est le **Profiler** qui s’occupe de gérer tout ça. On clique sur le nom de la route dans la barre d’outils ‘’oc\_platform\_home’’ si on est dans /platform.

Rappel sur les contrôleurs : OCPlatformBundle:Advert:view

Ici on indique que l’on utilise le Bundle OC/PlatformBundle, que l’on va chercher le contrôleur Advert et la méthode view.

## Les routes de base :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_home:

path: /platform

defaults:

\_controller: OCPlatformBundle:Advert:index

* Oc\_platform\_home n’a pas d’importance dans le fonctionnement, mais il doit être unique et pour être logique, respecter une certaine convention de nommage : oc\_platform en référence à OC/Platform.
* Path : /platform est l’URL sur laquelle la route s’applique.
* Defaults: \_controller : OCPlatformBundle :Advert :index est le paramètre de sortie de la route. Ici le seul contrôleur à appeler est mentionné, mais ce tableau defaults peut contenir d’autres paramètres.

### Créer une route avec des paramètres :

C’est la route 2 :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_view:

path: /platform/advert/{id}

defaults:

\_controller: OCPlatformBundle:Advert:view

Ici {id} prend n’importe quoi, ainsi la route sera sélectionnée pour : /platform/advert/5 par exemple. En revanche il doit être **OBLIGATOIREMENT PRESENT**.

Mais l’intérêt est d’avoir cet id en argument, on souhaite donc le récupérer.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

// La route fait appel à OCPlatformBundle:Advert:view,

// on doit donc définir la méthode viewAction.

// On donne à cette méthode l'argument $id, pour

// correspondre au paramètre {id} de la route

public function viewAction($id)

{

// $id vaut 5 si l'on a appelé l'URL /platform/advert/5

// Ici, on récupèrera depuis la base de données

// l'annonce correspondant à l'id $id.

// Puis on passera l'annonce à la vue pour

// qu'elle puisse l'afficher

return new *Response*("Affichage de l'annonce d'id : ".$id);

}

// ... et la méthode indexAction que nous avons déjà créée

}

En multipliant les paramètres on a comme route :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_view\_slug:

path: /platform/{year}/{slug}.{format}

defaults:

\_controller: OCPlatformBundle:Advert:viewSlug

Et aussi dans le contrôleur :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

// On récupère tous les paramètres en arguments de la méthode

public function viewSlugAction($slug, $year, $format)

{

return new *Response*(

"On pourrait afficher l'annonce correspondant au

slug '".$slug."', créée en ".$year." et au format ".$format."."

);

}

}

L’ordre n’a pas d’importance dans la déclaration de l’URL, ce qui compte c’est le nom des variables. On peut séparer les variables par un point ou un slash, les variables ne doivent donc pas contenir de point.

## Les routes avancées :

Pour éviter que l’utilisateur ne raconte trop de connerie, on va spécifier les types de variables que l’on attend :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_view\_slug:

path: /platform/{year}/{slug}.{format}

defaults:

\_controller: OCPlatformBundle:Advert:viewSlug

requirements:

year: \d{4}

format: html|xml

Avec requirements on spécifie ce qu’on attend dans year et format avec des expressions régulières.

### Paramètres facultatifs :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_view\_slug:

path: /platform/{year}/{slug}.{format}

defaults:

\_controller: OCPlatformBundle:Advert:viewSlug

format: html

requirements:

year: \d{4}

format: html|xml

avec l’ajout de format : on a un paramètre facultatif. On peut ne pas renseigner le format il sera par défaut HTML dans ce cas.

defaults: { \_controller: OCPlatformBundle:Advert:viewSlug, format: html }

(Si on souhaite écrire sur une seule ligne).

### Utiliser des paramètres systèmes :

Si on a XML comme format, on va donc afficher du XML et on veut donc envoyer le Header avec le bon Content-type. On va donc utiliser des paramètres systèmes et le Kernel de Symfony va effectuer des actions supplémentaires :

**\_format :**

Quand il est utilisé, un header avec le Content-type est ajouté à la réponse.

Il faut aussi penser à mettre $\_format dans viewSlugAction(). Si on fait l’exemple, le navigateur gueule parce que ce n’est pas un format XML valide et bien formé.

**\_locale :**

Définit la langue avec laquelle affiché la page si on dispose de la page en plusieurs langues.

Penser à mettre un requirements pour éviter que les utilisateurs ne demandent une langue que l’on ne maitrise pas.

**\_controller :**

On ne le mettra jamais dans le path, mais c’est bien un paramètre système.

### Ajouter un préfixe lors de l’import de nos routes :

Au lieu de mettre /platfom au début de chaque path, on peut rajouter un prefix :

Ce coup-ci on modifie le fichier app/config/routing.yml

# app/config/routing.yml

oc\_platform:

resource: "@OCPlatformBundle/Resources/config/routing.yml"

prefix: /platform

## Générer des URL :

### Pourquoi générer des URL ?

Le routeur peut associer une route à une certaine URL mais aussi reconstruire l’URL à une certaine route.

Exemple :

On a une route nommée ‘’oc\_platform\_view’’ qui écoute l’URL /platform/advert/{id}, on peut décider de racourcir l’URL pour n’avoir que /platform/a/{id}.

### Comment générer des URL :

**Depuis le contrôleur :**

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

public function indexAction()

{

// On veut avoir l'URL de l'annonce d'id 5.

$url = $this->get('router')->generate(

'oc\_platform\_view', // 1er argument : le nom de la route

array('id' => 5) // 2e argument : les valeurs des paramètres

);

// $url vaut « /platform/advert/5 »

return new *Response*("L'URL de l'annonce d'id 5 est : ".$url);

}

}

Le nom de la route ici est référencé dans le Routeur :

oc\_platform\_view:

path: /advert/{id}

defaults: { \_controller: OCPlatformBundle:Advert:view }

Le deuxième argument est facultatif si on n’utilise pas d’arguments.

La méthode nous renverra donc : l'URL de l'annonce d'id est : /Symfony/web/app\_dev.php/platform/advert/5.

On récupère l’URL pour l’afficher.

**Si on veut une URL absolue :**

On doit définir le troisième argument de la méthode generate :

<?php

use Symfony\Component\Routing\Generator\UrlGeneratorInterface;

$url = $this->get('router')->generate('oc\_platform\_home', array(), *UrlGeneratorInterface*::ABSOLUTE\_URL);

Ainsi on aura l’URL entière et pas uniquement la fin.

Comme notre contrôleur hérite de du contrôleur de Symfony on a accès à eux autres méthodes :

<?php

// Depuis un contrôleur

// Méthode longue

$url = $this->get('router')->generate('oc\_platform\_home');

// Méthode courte

$url = $this->generateUrl('oc\_platform\_home');

**Depuis une vue Twig :**

 On utilise path.

{# Dans une vue Twig, en considérant bien sûr

que la variable advert\_id est disponible #}

<a href="{{ path('oc\_platform\_view', { 'id': advert\_id }) }}">

Lien vers l'annonce d'id {{ advert\_id }}

</a>

Pour générer l’URL complète, on utilise url() à la place de path(). Pas besoin de troisème argument.

## Application : les routes de notre plateforme :

**Page d’accueil** : on veut pouvoir regarder les dernières annonces mais aussi les plus anciennes. On va donc définir {page}comme étant facultatif.

|  |  |
| --- | --- |
| /platform | page = 1 |
| /platform/1 | page = 1 |
| /platform/2 | page = 2 |

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_home:

path: /{page}

defaults:

\_controller: OCPlatformBundle:Advert:index

page: 1

requirements:

page: \d\*

**Page de visualisation d’une annonce :**

On rajoute juste un {id} pour choisir la bonne annonce.

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_view:

path: /advert/{id}

defaults:

\_controller: OCPlatformBundle:Advert:view

requirements:

id: \d+

**Ajout, modification et suppression :**

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_add:

path: /add

defaults:

\_controller: OCPlatformBundle:Advert:add

oc\_platform\_edit:

path: /edit/{id}

defaults:

\_controller: OCPlatformBundle:Advert:edit

requirements:

id: \d+

oc\_platform\_delete:

path: /delete/{id}

defaults:

\_controller: OCPlatformBundle:Advert:delete

requirements:

id: \d+

# Les contrôleurs avec Symfony

Le chef d’orchestre, doit renvoyer une réponse !

## Le rôle du contrôleur :

**Retourner une réponse :**

Symfony s’inspire des concepts http. Il existe donc dans Symfony les classes *Request* et *Response*. Renvoyer une réponse = instancier une de ces classes.

Comme $response et faire un return $response.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

public function indexAction()

{

return new *Response*("Hello World !");

}

}

## L’objet Request :

### Les paramètres contenus dans les routes :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_view:

path: /advert/{id}

defaults:

\_controller: OCPlatformBundle:Advert:view

requirements:

id: \d+

Le paramètre {id} est récupéré par la route et transformé en argument $id pour le contrôleur. Voici le code du contrôleur :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

// …

public function viewAction($id)

{

return new *Response*("Affichage de l'annonce d'id : ".$id);

}

}

### Les paramètres hors routes :

C’est la méthode « à l’ancienne », imaginons qu’on ait : /platform/advert/5?tag=developer et qu’on aimerait récupérer la variable tag. On utilise l’objet *Request*.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request; // N'oubliez pas ce use !

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

public function viewAction($id, Request $request)

{

// Vous avez accès à la requête HTTP via $request

}

}

On va d’abords récupérer la requête. L’action est réalisée par le Kernel, il regarde si l’un des arguments de la méthode est typé avec Request, si c’est le cas il ajoute la requête aux arguments avant d’exécuter le contrôleur.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

// …

// On injecte la requête dans les arguments de la méthode

public function viewAction($id, Request $request)

{

// On récupère notre paramètre tag

$tag = $request->query->get('tag');

return new *Response*(

"Affichage de l'annonce d'id : ".$id.", avec le tag : ".$tag

);

}

}

On a utilisé $request->query pour récupérer les paramètres de l’URL passés en GET, mais il y en a d’autre :

| **Type de paramètres** | **Méthode Symfony** | **Méthode traditionnelle** | **Exemple** |
| --- | --- | --- | --- |
| Variables d'URL | $request->query | $\_GET | $request->query->get('tag') |
| Variables de formulaire | $request->request | $\_POST | $request->request->get('tag') |
| Variables de cookie | $request->cookies | $\_COOKIE | $request->cookies->get('tag') |
| Variables de serveur | $request->server | $\_SERVER | $request->server->get('REQUEST\_URI') |
| Variables d'entête | $request->headers | $\_SERVER['HTTP\_\*'] | $request->headers->get('USER\_AGENT') |
| Paramètres de route | $request->attributes | n/a | On utilise $id dans les arguments de la méthode, mais vous pourriez également écrire  $request->attributes->get('id') |

Si on fait $request->query->get(‘sdf’) et que le paramètre sdf n’est pas dans l’URL, on aura une chaîne vide au lieu d’une erreur.

### Les autres méthodes de l’objet Request :

**Récupérer la méthode de la requête http :**

<?php

if ($request->isMethod('POST'))

{

// Un formulaire a été envoyé, on peut le traiter ici

}

**Savoir si la requête est une requête AJAX :**

<?php

if ($request->isXmlHttpRequest())

{

// C'est une requête AJAX, retournons du JSON, par exemple

}

**Les autres :**

[Sur le site de Symfony](http://api.symfony.com/3.0/Symfony/Component/HttpFoundation/Request.html) :

**Décomposition de la construction d’un objet Response :**

Cas d’une erreur 404 :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

// On modifie viewAction, car elle existe déjà

public function viewAction($id)

{

// On crée la réponse sans lui donner de contenu pour le moment

$response = new *Response*();

// On définit le contenu

$response->setContent("Ceci est une page d'erreur 404");

// On définit le code HTTP à « Not Found » (erreur 404)

$response->setStatusCode(*Response*::HTTP\_NOT\_FOUND);

// On retourne la réponse

return $response;

}

}

### Réponses et vues :

Templating dispose d’un raccourci : renderResponse(). Elle prend en paramètre le nom du template et ses variables.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

public function viewAction($id, Request $request)

{

// On récupère notre paramètre tag

$tag = $request->query->get('tag');

// On utilise le raccourci : il crée un objet Response

// Et lui donne comme contenu le contenu du template

return $this->get('templating')->renderResponse(

'OCPlatformBundle:Advert:view.html.twig',

array('id' => $id, 'tag' => $tag)

);

}

}

Ou alors avec uniquement render() dont dispose le contrôleur comme raccourci :

<?php

public function viewAction($id, Request $request)

{

// On récupère notre paramètre tag

$tag = $request->query->get('tag');

return $this->render('OCPlatformBundle:Advert:view.html.twig', array(

'id' => $id,

'tag' => $tag,

));

}

C’est ainsi qu’on gère la plupart des réponses, on ne manipule jamais directement les views.

Il faut bien sûr créer view.html.twig

{# src/OC/PlatformBundle/Resources/view/Advert/view.html.twig #}

<!DOCTYPE html>

<html>

<head>

<title>Affichage de l'annonce {{ id }}</title>

</head>

<body>

<h1>Hello Annonce n°{{ id }} !</h1>

<p>Tag éventuel : {{ tag }}</p>

</body>

</html>

### Réponse et redirection :

**Une redirection est une réponse http**. Il existe l’objet **RedirectResponse** qui étend l’objet Response en ajoutant l’entête **Location**.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\RedirectResponse; // N'oubliez pas ce use

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

public function viewAction($id)

{

$url = $this->get('router')->generate('oc\_platform\_home');

return new *RedirectResponse*($url);

}

}

Pour faire une redirection depuis un contrôleur : **redirect** : inutile d’intégrer le use RedirectResponse.

<?php

public function viewAction($id)

{

$url = $this->get('router')->generate('oc\_platform\_home');

return $this->redirect($url);

}

Ou de manière raccourcie :

<?php

public function viewAction($id)

{

return $this->redirectToRoute('oc\_platform\_home');

}

Ici on prend la route et non l’URL en argument. Pratique.

En passant intercept\_redirect à true dans app/config/config\_dev.yml. On peut intercepter les redirections.

### Changer le Content-type de la réponse :

Exemple : une requête AJAX qui renvoie un tableau au format JSON :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Response;

class AdvertController extends Controller

{

public function viewAction($id)

{

// Créons nous-mêmes la réponse en JSON, grâce à la fonction json\_encode()

$response = new *Response*(json\_encode(array('id' => $id)));

// Ici, nous définissons le Content-type pour dire au navigateur

// que l'on renvoie du JSON et non du HTML

$response->headers->set('Content-Type', 'application/json');

return $response;

}

De manière raccourci :

<?php

use Symfony\Component\HttpFoundation\JsonResponse;

// ...

public function viewAction($id)

{

return new *JsonResponse*(array('id' => $id));

}

## Manipuler la session :

On les défini et récupère avec GET et SET :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpFoundation\Response;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class AdvertController extends Controller

{

public function viewAction($id, Request $request)

{

// Récupération de la session

$session = $request->getSession();

// On récupère le contenu de la variable user\_id

$userId = $session->get('user\_id');

// On définit une nouvelle valeur pour cette variable user\_id

$session->set('user\_id', 91);

// On n'oublie pas de renvoyer une réponse

return new *Response*("<body>Je suis une page de test, je n'ai rien à dire</body>");

}

}

Dans le Profiler on peut voir la session et ses paramètres.

Il y a aussi une mémoire flash qui ne dure que le temps d’une page.

Exemple :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Component\HttpFoundation\Response;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class AdvertController extends Controller

{

public function viewAction($id)

{

return $this->render('OCPlatformBundle:Advert:view.html.twig', array(

'id' => $id

));

}

// Ajoutez cette méthode :

public function addAction(Request $request)

{

$session = $request->getSession();

// Bien sûr, cette méthode devra réellement ajouter l'annonce

// Mais faisons comme si c'était le cas

$session->getFlashBag()->add('info', 'Annonce bien enregistrée');

// Le « flashBag » est ce qui contient les messages flash dans la session

// Il peut bien sûr contenir plusieurs messages :

$session->getFlashBag()->add('info', 'Oui oui, elle est bien enregistrée !');

// Puis on redirige vers la page de visualisation de cette annonce

return $this->redirectToRoute('oc\_platform\_view', array('id' => 5));

}

}

Le message flash s’affiche puis est détruit par la session.

La lecture des messages se fait dans la vue :

{# src/OC/PlatformBundle/Resources/view/Advert/view.html.twig #}

<!DOCTYPE html>

<html>

<head>

<title>Affichage de l'annonce {{ id }}</title>

</head>

<body>

<h1>Affichage de l'annonce n°{{ id }} !</h1>

<div>

{# On affiche tous les messages flash dont le nom est « info » #}

{% for message in app.session.flashbag.get('info') %}

<p>Message flash : {{ message }}</p>

{% endfor %}

</div>

<p>

Ici nous pourrons lire l'annonce ayant comme id : {{ id }}<br />

Mais pour l'instant, nous ne savons pas encore le faire, cela viendra !

</p>

</body>

</html>

En allant à [platform/add](http://localhost/Symfony/web/app_dev.php/platform/add) on est redirigé sur la page Avert/5 et on a bien les messages flashs.

## Le contrôleur de note plateforme :

Pour le mêment :

# src/OC/PlatformBundle/Resources/config/routing.yml

oc\_platform\_home:

path: /{page}

defaults:

\_controller: OCPlatformBundle:Advert:index

page: 1

requirements:

page: \d\*

oc\_platform\_view:

path: /advert/{id}

defaults:

\_controller: OCPlatformBundle:Advert:view

requirements:

id: \d+

oc\_platform\_add:

path: /add

defaults:

\_controller: OCPlatformBundle:Advert:add

oc\_platform\_edit:

path: /edit/{id}

defaults:

\_controller: OCPlatformBundle:Advert:edit

requirements:

id: \d+

oc\_platform\_delete:

path: /delete/{id}

defaults:

\_controller: OCPlatformBundle:Advert:delete

requirements:

id: \d+

Et le contrôleur Advert :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class AdvertController extends Controller

{

public function indexAction($page)

{

// On ne sait pas combien de pages il y a

// Mais on sait qu'une page doit être supérieure ou égale à 1

if ($page < 1) {

// On déclenche une exception NotFoundHttpException, cela va afficher

// une page d'erreur 404 (qu'on pourra personnaliser plus tard d'ailleurs)

throw new *NotFoundHttpException*('Page "'.$page.'" inexistante.');

}

// Ici, on récupérera la liste des annonces, puis on la passera au template

// Mais pour l'instant, on ne fait qu'appeler le template

return $this->render('OCPlatformBundle:Advert:index.html.twig');

}

public function viewAction($id)

{

// Ici, on récupérera l'annonce correspondante à l'id $id

return $this->render('OCPlatformBundle:Advert:view.html.twig', array(

'id' => $id

));

}

public function addAction(Request $request)

{

// La gestion d'un formulaire est particulière, mais l'idée est la suivante :

// Si la requête est en POST, c'est que le visiteur a soumis le formulaire

if ($request->isMethod('POST')) {

// Ici, on s'occupera de la création et de la gestion du formulaire

$request->getSession()->getFlashBag()->add('notice', 'Annonce bien enregistrée.');

// Puis on redirige vers la page de visualisation de cettte annonce

return $this->redirectToRoute('oc\_platform\_view', array('id' => 5));

}

// Si on n'est pas en POST, alors on affiche le formulaire

return $this->render('OCPlatformBundle:Advert:add.html.twig');

}

public function editAction($id, Request $request)

{

// Ici, on récupérera l'annonce correspondante à $id

// Même mécanisme que pour l'ajout

if ($request->isMethod('POST')) {

$request->getSession()->getFlashBag()->add('notice', 'Annonce bien modifiée.');

return $this->redirectToRoute('oc\_platform\_view', array('id' => 5));

}

return $this->render('OCPlatformBundle:Advert:edit.html.twig');

}

public function deleteAction($id)

{

// Ici, on récupérera l'annonce correspondant à $id

// Ici, on gérera la suppression de l'annonce en question

return $this->render('OCPlatformBundle:Advert:delete.html.twig');

}

}

# Le moteur de template Twig

## Les templates Twig :

### Intérêt :

Permettent de séparer le code HTML du code XML, texte … Il sécurise les variables, il y a des fonctionnalités en plus.

### Page web, email …

Les templates doivent être utilisés partout, email, flux RSS …

### En pratique :

<?php

// Depuis un contrôleur

return $this->render('OCPlatformBundle:Advert:index.html.twig', array(

'var1' => $var1,

'var2' => $var2

));

Et voici comment, au milieu d'un contrôleur, récupérer le contenu d'un template en texte :

<?php

// Depuis un contrôleur

$contenu = $this->renderView('OCPlatformBundle:Advert:email.txt.twig', array(

'var1' => $var1,

'var2' => $var2

));

// Puis on envoie l'e-mail, par exemple :

mail('moi@openclassrooms.com', 'Inscription OK', $contenu);

Et le template OCPlatformBundle:Advert:email.txt.twig contiendrait par exemple :

{# src/OC/PlatformBundle/Resources/views/Advert/email.txt.twig #}

Bonjour {{ pseudo }},

Toute l'équipe du site se joint à moi pour vous souhaiter

la bienvenue sur notre site !

Revenez nous voir souvent !

**A savoir :**

* {{ … }}*affiche* quelque chose ;
* {% … %}*fait* quelque chose ;
* {# … #}  n'affiche rien et ne fait rien : c'est la syntaxe pour les commentaires, qui peuvent être sur plusieurs lignes.

## Afficher des variables :

| **Description** | **Exemple Twig** | **Équivalent PHP** |
| --- | --- | --- |
| Afficher une variable | Pseudo : {{ pseudo }} | Pseudo : <?php echo $pseudo; ?> |
| Afficher l'index d'un tableau | Identifiant : {{ user['id'] }} | Identifiant : <?php echo $user['id']; ?> |
| Afficher l'attribut d'un objet, dont le getter respecte la convention $objet->getAttribut() | Identifiant : {{ user.id }} | Identifiant : <?php echo $user->getId(); ?> |
| Afficher une variable en lui appliquant un filtre. Ici, « upper » met tout en majuscules : | Pseudo en majuscules : {{ pseudo|upper }} | Pseudo en lettre majuscules : <?php echo strtoupper($pseudo); ?> |
| Afficher une variable en combinant les filtres. « striptags » supprime les balises HTML. « title » met la première lettre de chaque mot en majuscule. Notez l'ordre d'application des filtres, ici striptags est appliqué, puis title. | Message : {{ news.texte|striptags|title }} | Message : <?php echo ucwords(strip\_tags($news->getTexte())); ?> |
| Utiliser un filtre avec des arguments. Attention, il faut que date soit un objet de type Datetime ici. | Date : {{ date|date('d/m/Y') }} | Date : <?php echo $date->format('d/m/Y'); ?> |
| Concaténer | Identité : {{ nom ~ " " ~ prenom }} | Identité : <?php echo $nom.' '.$prenom; ?> |

**Quelques filtres :**

| **Filtre** | **Description** | **Exemple Twig** |
| --- | --- | --- |
| [upper](http://twig.sensiolabs.org/doc/filters/upper.html) | Met toutes les lettres en majuscules. | {{ var|upper }} |
| [striptags](http://twig.sensiolabs.org/doc/filters/striptags.html) | Supprime toutes les balises XML. | {{ var|striptags }} |
| [date](http://twig.sensiolabs.org/doc/filters/date.html) | Formate la date selon le format donné en argument. La variable en entrée doit être une instance de Datetime. | {{ date|date('d/m/Y') }} Date d'aujourd'hui : {{ "now"|date('d/m/Y') }} |
| [format](http://twig.sensiolabs.org/doc/filters/format.html) | Insère des variables dans un texte, équivalent à [printf](https://php.net/printf). | {{ "Il y a %s pommes et %s poires"|format(153, nb\_poires) }} |
| [length](http://twig.sensiolabs.org/doc/filters/length.html) | Retourne le nombre d'éléments du tableau, ou le nombre de caractères d'une chaîne. | Longueur de la variable : {{ texte|length }} Nombre d'éléments du tableau : {{ tableau|length }} |

### Twig et la sécurité :

Twig protège automatiquement les variables. Pour ne plus échapper les balises <> de l’HTML, on utilise RAW : {{ ma\_variable\_html|raw }}

### Les variables globales :

Twig créer automatiquement {{ app }}, on a donc accès à :

| **Variable** | **Description** |
| --- | --- |
| {{ app.request }} | La requête « request » qu'on a vue au chapitre précédent sur les contrôleurs. |
| {{ app.session }} | Le service « session » qu'on a vu également au chapitre précédent. |
| {{ app.environment }} | L'environnement courant : « dev », « prod », et ceux que vous avez définis. |
| {{ app.debug }} | True si le mode debug est activé, False sinon. |
| {{ app.user }} | L'utilisateur courant, que nous verrons également plus loin dans ce cours. |

Pour éditer ses propres variables globales, on peut faire ainsi :

# app/config/config.yml

# …

twig:

# …

globals:

webmaster: moi-même

Ainsi {{ webmaster }} sera accessible dans toutes les vues.

Je profite de cet exemple pour vous faire passer un petit message. Pour ce genre de valeurs paramétrables, la bonne pratique est de les définir non pas directement dans le fichier de configuration config.yml, mais dans le fichier des paramètres, à savoir parameters.yml. Attention, je parle bien de la valeur du paramètre, non de la configuration. Voyez par vous-mêmes.

Valeur du paramètre :

# app/config/parameters.yml

parameters:

# …

app\_webmaster: moi-même

Configuration (ici, injection dans toutes les vues) qui utilise le paramètre :

# app/config/config.yml

twig:

globals:

webmaster: %app\_webmaster%

On a ainsi séparé la valeur du paramètre, stockée dans un fichier simple, de l'utilisation de ce paramètre, perdue dans le fichier de configuration.

## Structure et contrôle des expressions :

### Les structures de contrôle :

##### Condition : [{% if %}](http://twig.sensiolabs.org/doc/tags/if.html)

Exemple Twig :

{% if membre.age < 12 %}

Il faut avoir au moins 12 ans pour ce film.

{% elseif membre.age < 18 %}

OK bon film.

{% else %}

Un peu vieux pour voir ce film non ?

{% endif %}

Équivalent PHP :

<?php if($membre->getAge() < 12) { ?>

Il faut avoir au moins 12 ans pour ce film.

<?php } elseif($membre->getAge() < 18) { ?>

OK bon film.

<?php } else { ?>

Un peux vieux pour voir ce film non ?

<?php } ?>

##### Boucle : [{% for %}](http://twig.sensiolabs.org/doc/tags/for.html)

Exemple Twig :

<ul>

{% for membre in liste\_membres %}

<li>{{ membre.pseudo }}</li>

{% else %}

<li>Pas d'utilisateur trouvé.</li>

{% endfor %}

</ul>

Et pour avoir accès aux clés du tableau :

<select>

{% for valeur, option in liste\_options %}

<option value="{{ valeur }}">{{ option }}</option>

{% endfor %}

</select>

Équivalent PHP :

<ul>

<?php if(count($liste\_membres) > 0) {

foreach($liste\_membres as $membre) {

echo '<li>'.$membre->getPseudo().'</li>';

}

} else { ?>

<li>Pas d'utilisateur trouvé.</li>

<?php } ?>

</ul>

Avec les clés :

<?php

foreach($liste\_options as $valeur => $option) {

// …

}

##### Définition : [{% set %}](http://twig.sensiolabs.org/doc/tags/set.html)

Exemple Twig :

{% set foo = 'bar' %}

Équivalent PHP :

<?php $foo = 'bar'; ?>

Une petite information sur la structure {% for %}, celle-ci définit une variable {{ loop }} au sein de la boucle, qui contient les attributs suivants :

| **Variable** | **Description** |
| --- | --- |
| {{ loop.index }} | Le numéro de l'itération courante (en commençant par 1). |
| {{ loop.index0 }} | Le numéro de l'itération courante (en commençant par 0). |
| {{ loop.revindex }} | Le nombre d'itérations restantes avant la fin de la boucle (en finissant par 1). |
| {{ loop.revindex0 }} | Le nombre d'itérations restantes avant la fin de la boucle (en finissant par 0). |
| {{ loop.first }} | true si c'est la première itération, false sinon. |
| {{ loop.last }} | true si c'est la dernière itération, false sinon. |
| {{ loop.length }} | Le nombre total d'itérations dans la boucle. |

#### Les tests utiles

##### [Defined](http://twig.sensiolabs.org/doc/tests/defined.html)

Pour vérifier si une variable existe.

Exemple Twig :

{% if var is defined %} … {% endif %}

Équivalent PHP :

<?php if(isset($var)) { }

##### [Even](http://twig.sensiolabs.org/doc/tests/even.html) / [Odd](http://twig.sensiolabs.org/doc/tests/odd.html)

Pour tester si un nombre est pair / impair.

Exemple Twig :

{% for valeur in liste %}

<span class="{% if loop.index is even %}pair{% else %}

impair{% endif %}">

{{ valeur }}

</span>

{% endfor %}

Équivalent PHP :

<?php

$i = 0;

foreach($liste as $valeur) {

echo '<span class="';

echo $i % 2 ? 'impair' : 'pair';

echo '">'.$valeur.'</span>';

$i++;

}

## Hériter et inclure des templates :

Problème : un seul design mais on ne veut pas le répéter dans chaque template.

Pour résoudre le problème, on a un design père avec des blocs « trous », on définit ensuite dans des classes fils, des designs pour remplir les trous.

### La pratique :

Les templates pères sont appelés layout :

{# src/OC/PlatformBundle/Resources/views/layout.html.twig #}

<!DOCTYPE HTML>

<html>

<head>

<meta charset="utf-8">

<title>{% block title %}OC Plateforme{% endblock %}</title>

</head>

<body>

{% block body %}

{% endblock %}

</body>

</html>

Template fils :

src/OC/PlatformBundle/Resources/views/Advert/index.html.twig :

{# src/OC/PlatformBundle/Resources/views/Advert/index.html.twig #}

{% extends "OCPlatformBundle::layout.html.twig" %}

{% block title %}{{ parent() }} - Index{% endblock %}

{% block body %}

Notre plateforme est un peu vide pour le moment, mais cela viendra !

{% endblock %}

### Le nom du template père :

Pour la notation, on ne fait pas OCPlatformBundle :Controleur :layout.html.twig mais plutôt OCPlatformBundle ::layout.html.twig.

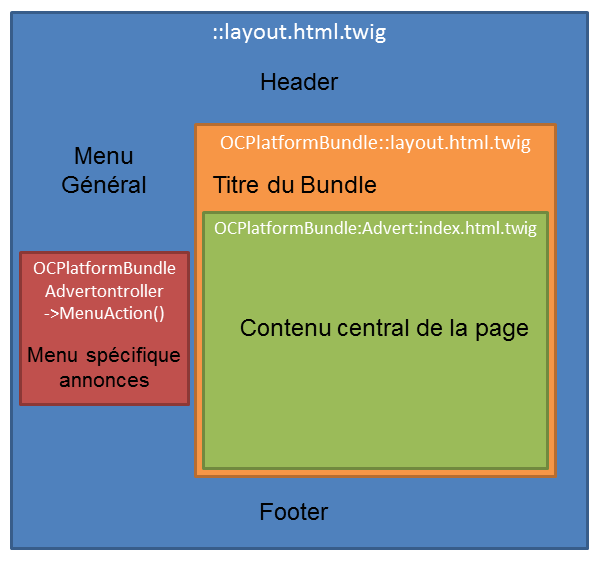
On peut mettre une valeur par défaut comme on a fait pour le titre.

* On peut récupérer cette valeur avec {{ parent() }}
* Si on ne redéfinit pas, on aura toujours une valeur.

### Organiser son code en triple héritage :

* Layout général : c'est le design de votre site, indépendamment de vos bundles. Il contient l'en-tête, le pied de page, etc. La structure de votre site donc (c'est notre template père).
* Layout du bundle : il hérite du layout général et contient les parties communes à toutes les pages d'un même bundle. Par exemple, pour notre plateforme d'annonce, on pourrait afficher un menu particulier, rajouter « Annonces » dans le titre, etc.
* Template de page : il hérite du layout du bundle et contient le contenu central de votre page.

Le layout général se met dans App. app/Resources/views/layout.html.twig



### Héritage ou inclusion ?

Ajouter et modifier une annonce utilisent le même formulaire. On ne va pas faire de copier-coller du code, on utilise l’inclusion : On a nos deux templates OCPlatformBundle:Advert:add.html.twig et OCPlatformBundle:Advert:edit.html.twig qui héritent chacun de OCPlatformBundle::layout.html.twig.

L'affichage exact de ces deux templates diffère un peu, mais chacun d'eux inclut OCPlatformBundle:Advert:form.html.twig à l'endroit exact pour afficher le formulaire.

### La pratique :

L’include ( ):

{# src/OC/PlatformBundle/Resources/views/Advert/add.html.twig #}

{% extends "OCPlatformBundle::layout.html.twig" %}

{% block body %}

<h2>Ajouter une annonce</h2>

{{ include("OCPlatformBundle:Advert:form.html.twig") }}

<p>

Attention : cette annonce sera ajoutée directement

sur la page d'accueil après validation du formulaire.

</p>

{% endblock %}

Code du template inclus :

{# src/OC/PlatformBundle/Resources/views/Advert/form.html.twig #}

{# Cette vue n'hérite de personne, elle sera incluse par d'autres vues qui,

elles, hériteront probablement du layout. Je dis « probablement » car,

ici pour cette vue, on n'en sait rien et c'est une info qui ne nous concerne pas. #}

<h3>Formulaire d'annonce</h3>

{# On laisse vide la vue pour l'instant, on la comblera plus tard

lorsqu'on saura afficher un formulaire. #}

<div class="well">

Ici se trouvera le formulaire.

</div>

### Inclusion de contrôleur :

Pour inclure un contrôleur on utilise {{ render() }}. Exemple avec le layout :

{# src/OC/PlatformBundle/Resources/views/layout.html.twig #}

<!DOCTYPE HTML>

<html>

<head>

<meta charset="utf-8">

<title>{% block title %}OC Plateforme{% endblock %}</title>

</head>

<body>

<div id="menu">

{{ render(controller("OCPlatformBundle:Advert:menu")) }}

</div>

{% block body %}

{% endblock %}

</body>

</html>

On inclut un contrôleur quand on a besoin de quelque chose de précis qu’un template ne peut pas nous donner tout seul. On inclut le contrôleur qui génère les paramètres et appel ç son tour le template.

Il faut aussi ajouter la méthode menuAction dans le contrôleur.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class AdvertController extends Controller

{

// ...

public function menuAction()

{

// On fixe en dur une liste ici, bien entendu par la suite

// on la récupérera depuis la BDD !

$listAdverts = array(

array('id' => 2, 'title' => 'Recherche développeur Symfony'),

array('id' => 5, 'title' => 'Mission de webmaster'),

array('id' => 9, 'title' => 'Offre de stage webdesigner')

);

return $this->render('OCPlatformBundle:Advert:menu.html.twig', array(

// Tout l'intérêt est ici : le contrôleur passe

// les variables nécessaires au template !

'listAdverts' => $listAdverts

));

}

}

Exemple de ce que pourrait être un template :

{# src/OC/PlatformBundle/Resources/views/Advert/menu.html.twig #}

{# Ce template n'hérite de personne,

tout comme le template inclus avec {{ include() }}. #}

<ul class="nav nav-pills nav-stacked">

{% for advert in listAdverts %}

<li>

<a href="{{ path('oc\_platform\_view', {'id': advert.id}) }}">

{{ advert.title }}

</a>

</li>

{% endfor %}

</ul>

## Application :

On travaille avec des variables vides au lieu de BDD pour le moment.

### Layout général :

{# app/Resources/views/layout.html.twig #}

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<meta name="viewport" content="width=device-width, initial-scale=1">

<title>{% block title %}OC Plateforme{% endblock %}</title>

{% block stylesheets %}

{# On charge le CSS de bootstrap depuis le site directement #}

<link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/css/bootstrap.min.css">

{% endblock %}

</head>

<body>

<div class="container">

<div id="header" class="jumbotron">

<h1>Ma plateforme d'annonces</h1>

<p>

Ce projet est propulsé par Symfony,

et construit grâce au MOOC OpenClassrooms et SensioLabs.

</p>

<p>

<a class="btn btn-primary btn-lg" href="https://openclassrooms.com/courses/developpez-votre-site-web-avec-le-framework-symfony2">

Participer au MOOC »

</a>

</p>

</div>

<div class="row">

<div id="menu" class="col-md-3">

<h3>Les annonces</h3>

<ul class="nav nav-pills nav-stacked">

<li><a href="{{ path('oc\_platform\_home') }}">Accueil</a></li>

<li><a href="{{ path('oc\_platform\_add') }}">Ajouter une annonce</a></li>

</ul>

<h4>Dernières annonces</h4>

{{ render(controller("OCPlatformBundle:Advert:menu", {'limit': 3})) }}

</div>

<div id="content" class="col-md-9">

{% block body %}

{% endblock %}

</div>

</div>

<hr>

<footer>

<p>The sky's the limit © {{ 'now'|date('Y') }} and beyond.</p>

</footer>

</div>

{% block javascripts %}

{# Ajoutez ces lignes JavaScript si vous comptez vous servir des fonctionnalités du bootstrap Twitter #}

<script src="//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js"></script>

<script src="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.min.js"></script>

{% endblock %}

</body>

</html>

### Layout du Bundle :

{# src/OC/PlatformBundle/Resources/views/layout.html.twig #}

{% extends "::layout.html.twig" %}

{% block title %}

Annonces - {{ parent() }}

{% endblock %}

{% block body %}

{# On définit un sous-titre commun à toutes les pages du bundle, par exemple #}

<h1>Annonces</h1>

<hr>

{# On définit un nouveau bloc, que les vues du bundle pourront remplir #}

{% block ocplatform\_body %}

{% endblock %}

{% endblock %}

### Templates finaux :

{# src/OC/PlatformBundle/Resources/views/Advert/index.html.twig #}

{% extends "OCPlatformBundle::layout.html.twig" %}

{% block title %}

Accueil - {{ parent() }}

{% endblock %}

{% block ocplatform\_body %}

<h2>Liste des annonces</h2>

<ul>

{% for advert in listAdverts %}

<li>

<a href="{{ path('oc\_platform\_view', {'id': advert.id}) }}">

{{ advert.title }}

</a>

par {{ advert.author }},

le {{ advert.date|date('d/m/Y') }}

</li>

{% else %}

<li>Pas (encore !) d'annonces</li>

{% endfor %}

</ul>

{% endblock %}

La suite de notre appli dans le cours sur Twig.

# Installer un Bundle avec Composer

## C’est quoi ?

Il gère les dépendances. Exemple, si on utilise SwiftMailer pour envoyer nos mail, SwiftMailer est une dépendance de notre projet.

Les problèmes avec les bibliothèques externes et donc les dépendances :

* Ces bibliothèques sont mises à jour. Il vous faut donc les mettre à jour une à une pour vous assurer de corriger les bogues de chacune d'entre elles.
* Ces bibliothèques peuvent elles-mêmes dépendre d'autres bibliothèques. En effet, si une de vos bibliothèques dépend d'autres bibliothèques, cela vous oblige à gérer l'ensemble de ces dépendances (installation, mises à jour, etc.).
* Ces bibliothèques ont chacune leur paramètres d'*autoload*, et vous devez gérer leur *autoload* pour chacune d'entre elles.

Pour que Composer sache où sont nos bibliothèques : si on regarde sur la bibliothèque de Symfony [ici](https://packagist.org/packages/symfony/symfony), on voit dans Require tout ce dont on a besoin pour le faire fonctionner.

Pour savoir comment ça fonctionne :

* On définit dans un fichier la liste des bibliothèques dont le projet dépend, ainsi que leur version ;
* On exécute une commande pour installer ou mettre à jour ces bibliothèques (et leurs propres dépendances donc) ;
* On inclut alors le fichier d'*autoload* généré par Composer dans notre projet.

## L’installation :

#### Installer Composer

Installer Composer est très facile, il suffit d'une seule commande… PHP ! Exécutez cette commande dans la console :

C:\wamp\www> php -r "eval('?>'.file\_get\_contents('http://getcomposer.org/installer'));"

Cette commande télécharge et exécute le fichier PHP suivant : [http://getcomposer.org/installer](https://getcomposer.org/installer) . Vous pouvez aller le voir, ce n'est pas Composer en lui-même mais son installateur. Il fait quelques vérifications (version de PHP, etc.), puis télécharge effectivement Composer dans le fichier composer.phar.

Composer en lui-même, c'est le fichier PHAR composer.phar, c'est lui que nous devrons exécuter par la suite. Vous pouvez déjà l'exécuter pour vérifier que tout est OK :

C:\wamp\www>php composer.phar --version

Composer version a5eaba8

N'hésitez pas à mettre à jour Composer lui-même de temps en temps. Il faut pour cela utiliser la commande self-update de Composer, comme suit :

C:\wamp\www>php composer.phar self-update

Updating to version ded485d.

    Downloading: 100%

Et voilà, je viens de mettre à jour Composer très simplement !

Cependant, l'installation n'est pas finie. En effet, pour récupérer certaines bibliothèques, Composer utilise Git.

## Installer un Bundle :

D’abord on créée un dossier test là où on a installé Composer. Dedans on créée un fichier json :

{

"require": {

"twig/extensions": "~1.0"

}

}

| **Valeur** | **Exemple** | **Description** |
| --- | --- | --- |
| Un numéro de version exact | "2.0.17" | Ainsi, Composer téléchargera cette version exacte. |
| Une plage de versions | ">=2.0,<2.6" | Ainsi, Composer téléchargera la version la plus à jour, à partir de la version 2.0 et en s'arrêtant avant la version 2.6. Par exemple, si les dernières versions sont 2.4, 2.5 et 2.6, Composer téléchargera la version 2.5. |
| Une plage de versions sémantique | "~2.1" | Ainsi, Composer téléchargera la version la plus à jour, à partir de la version 2.1 et en s'arrêtant avant la version 3.0. C'est une façon plus simple d'écrire ">=2.1,<3.0"  avec la syntaxe précédente. C'est la façon la plus utilisée pour définir la version des dépendances. |
| Un numéro de version avec joker « \* » | "2.0.\*" | Ainsi, Composer téléchargera la version la plus à jour qui commence par 2.0. Par exemple, il téléchargerait la version 2.0.17, mais pas la version 2.1.1. |
| Un nom de branche "dev-XXX" |  | C'est un cas un peu particulier, où Composer ira chercher la dernière modification d'une branche Git en particulier. N'utilisez cette syntaxe que pour les bibliothèques dont il n'existe pas de vraie version. Vous verrez assez souvent "dev-master", où "master" correspond à la branche principale d'un dépôt Git. |

Pour mettre à jour les dépendances on utilise update.

C:\wamp\www\test>php ../composer.phar update

Loading composer repositories with package information

Updating dependencies (including require-dev)

- Installing twig/twig (v1.23.1)

Downloading: 100%

- Installing twig/extensions (v1.3.0)

Downloading: 100%

Writing lock file

Generating autoload files

C:\wamp\www\test>

On peut vérifier dans le répertoire test/vendor :

* Composer a téléchargé la dépendance "twig/extensions" que l'on a définie, dans vendor/twig/extensions ;
* Composer a téléchargé la dépendance "twig/twig" de notre dépendance à nous, dans vendor/twig/twig ;
* Composer a généré les fichiers nécessaires pour l'*autoload*, allez vérifier le fichier vendor/composer/autoload\_namespaces.php.

### Installer un Bundle :

* On se rend [ici](https://packagist.org), ici on recherche ‘’fixture’’, et clique sur le Bundle de Doctrine : doctrine/doctrine-fixtures-bundle. On regarde les prérequis, ici on prend la version 2.3 qui peut prendre Symfony 2 et 3.
* Une fois que l’on connait le Bundle ET sa version, on modifie composer.json pour y rajouter :

"incenteev/composer-parameter-handler": "~2.0",

"doctrine/doctrine-fixtures-bundle": "~2.3"

* On remet à jour avec : php composer.phar update.

Enfin, on modifie AppKernel pour déclarer le Bundle :

<?php

// app/AppKernel.php

// …

if (in\_array($this->getEnvironment(), array('dev', 'test'))) {

// ...

$bundles[] = new *Doctrine*\Bundle\FixturesBundle\DoctrineFixturesBundle();

}

// …

Ici on indique que l’on veut le Bundle seulement en phase de dev et de test, si on le veut aussi en prod’ il faut le mettre en dehors du if.

Composer s’occupe de tout, même de déclarer les namespaces pour l’autoload. On peut vérifier dans vendor/composer/autoload\_namespace.php.

On y voit la ligne dans app/autoload.php :

<?php

$loader = require \_\_DIR\_\_.'/../vendor/autoload.php';

// …

### Gérer l’autoload :

On modifie le composer.json et pas les autres.

// composer.json

{

// …

"autoload": {

"psr-4": {

"": "src/",

"VotreNamespace": "chemin/vers/la/bibliotheque"

},

"files": [ "app/AppKernel.php" ]

},

// …

}

Ensuite il faut exécuter la commande :

C:\wamp\www\Symfony>php ../composer.phar dump-autoload

Generating autoload files

# Services, théories et créations

## Pourquoi utiliser des services ?

En PHP, on utilise beaucoup d’objets : envoyer des e-mail, BDD …

En générale, chaque objet utilise d’autres objets, comment organiser l’instanciation et par lequel commencer ?

Chaque objet est défini en **tant** que service et le **conteneur** permet d’instancier.

### Qu’est-ce qu’un service ?

Un service est un objet PHP qui remplir une fonction associée à une configuration. Il est accessible n’importe où dans le code. Pour chaque fonctionnalité on créée un ou plusieurs services. Un service est une **CLASSE**.

La configuration du service est un moyen de l’enregistrer dans le conteneur de service.

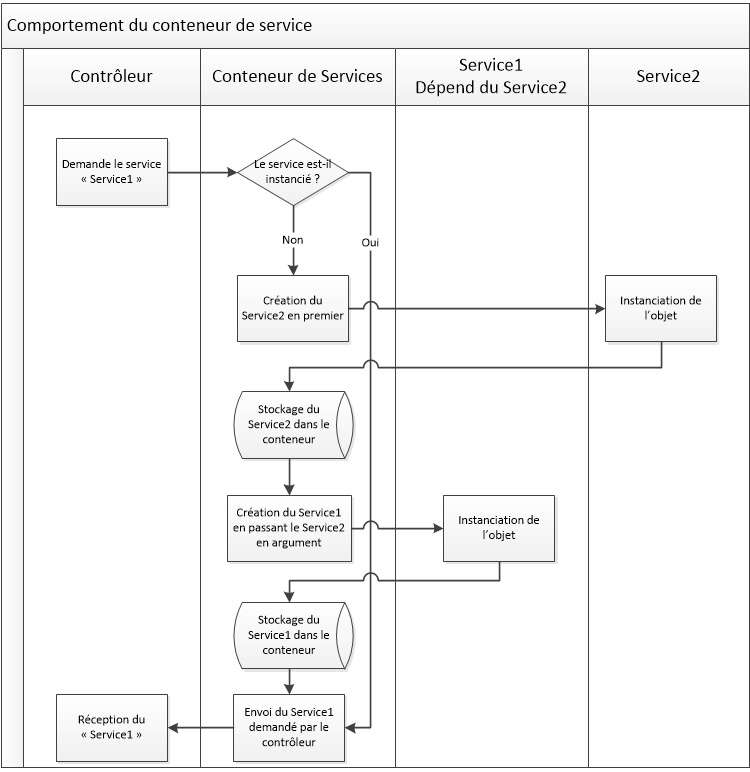
Prenons pour exemple l'envoi d'e-mails. Dans Symfony il existe le composant SwiftMailer qui permet de gérer les e-mails. Ce composant contient une classe nommée Swift\_Mailer qui envoie effectivement les e-mails. Symfony, qui intègre le composant SwiftMailer, définit déjà cette classe en tant que service mailer grâce à un peu de configuration. Le conteneur de service de Symfony peut donc accéder à la classe Swift\_Mailer  grâce au service mailer .

### Avantage de la POS (Programmation Orientée Service) :

Cela force à bien séparer chaque fonctionnalité. Comme chaque service ne remplit qu’une seule fonction, on peut facilement les réutiliser. On dit **Architecture Orienté Service**.

### Le conteneur des services :

C’est une sorte de super-objet qui gère tous les services. L’objectif est de simplifier au maximum la récupération des services. On l’appelle juste par son nom et le conteneur fait le reste.



Voilà ce qu’on aurait pu écrire dans ce code fictif si on avait dû écrire ce Container :

<?php

class Container

{

protected $service1 = null;

protected $service2 = null;

public function getService1()

{

if (null !== $this->service1) {

return $this->service1;

}

$service2 = $this->getService2();

$this->service1 = new *Service1*($service2);

return $this->service1;

}

public function getService2()

{

if (null !== $this->service2) {

return $this->service2;

}

$this->service2 = new *Service2*();

return $this->service2;

}

}

C’est comme ça que fait Symfony, preuve ici : var/cache/dev/appDevDebugProjectContainer.php

On a déjà utilisé un service de Templating, cette méthode a en fait :

<?php

protected function getTemplatingService()

{

$this->services['templating'] = $instance = new \Symfony\Bundle\TwigBundle\TwigEngine(

$this->get('twig'),

$this->get('templating.name\_parser'),

$this->get('templating.locator')

);

return $instance;

}

Ils se trouvent dans le cache, détruit à chaque fois donc. Comme ça si notre dépendance ajoute ou supprime une dépendance, le conteneur de service reflète le changement.

### Comment définir une dépendance entre service ?

L'idée est de définir pour chaque service :

* Son nom, qui permettra de l'identifier au sein du conteneur ;
* Sa classe, qui permettra au conteneur d'instancier le service ;
* Les arguments dont il a besoin. Un argument peut être un autre service, mais aussi un paramètre (défini dans le fichier parameters.yml par exemple).

### Le partage des services :

Dans Symfony, chaque service est partagé : la classe du service est instanciée une seule fois par le conteneur.

## Utiliser un service en pratique :

### Récupérer un service :

Le composant Swiftmailer permet d’envoyer des emails. Présent par défaut sous le service **Mailer**. On utilise simplement un **GET**.

<?php

$container->get('mailer');

Pour avoir la liste des services disponibles, utilisez la commande php bin/console debug:container.

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class AdvertController extends Controller

{

public function indexAction()

{

// On a donc accès au conteneur :

$mailer = $this->container->get('mailer');

// On peut envoyer des e-mails, etc.

}

}

On a déjà utilisé $this->get() c’est parce que la classe Controller permet d’utiliser un raccourci, au lieu d’utiliser $this->controller->get( ).

## Créer un service simple :

La seule convention à respecter dans Symfony, c’est de **mettre notre classe dans un namespace correspondant au dossier où est le fichier**.

Par exemple, la classeOC\PlatformBundle\Antispam\OCAntispam doit se trouver dans le répertoiresrc/OC/PlatformBundle/Antispam/OCAntispam.php

On va créer un système anti-spam, il pourra être utilisable dans d’autres projets, on va le créer dans un Bundle à part donc. Pour que ce soit un simple exemple, on le laisse dans notre Bundle.

<?php

// src/OC/PlatformBundle/Antispam/OCAntispam.php

namespace OC\PlatformBundle\Antispam;

class OCAntispam

{

/\*\*

\* Vérifie si le texte est un spam ou non

\*

\* @param string $text

\* @return bool

\*/

public function isSpam($text)

{

return strlen($text) < 50;

}

}

Rien besoin de plus pour créer un service. On a respecté le namespace. Ici on considère que c’est un spam au-delà de 50 caractères.

### Création de la configuration du service :

Il faut maintenant la signaler au conteneur de service. (DependencyInjection/OCPlatformExtension.php) devrait être créé automatiquement, sinon se référer au cours.

Dans services.yml de Ressources/config, on ajoute ou créée :

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.antispam:

class: OC\PlatformBundle\Antispam\OCAntispam

Dans cette configuration :

* oc\_platform.antispamest le nom de notre service fraîchement créé. De cette manière, le service sera accessible via $container->get('oc\_platform.antispam');. Essayez de respecter la convention en préfixant le nom de vos services par le nom du bundle, ici « oc\_platform ».
* class est un attribut obligatoire de notre configuration, il définit simplement le namespace complet de la classe du service. Cela indique au conteneur de services quelle classe instancier lorsqu'on lui demandera le service.

Pour définir un paramètre, la technique est la même que pour un service, dans le fichier services.yml:

parameters:

mon\_parametre: ma\_valeur

services:

# ...

Et pour accéder à ce paramètre, la technique est la même également, sauf qu'il faut utiliser la méthode $container->getParameter('nomParametre');au lieu deget(). C'est d'ailleurs comme cela que vous pouvez récupérer les paramètres qui sont dans le fichier app/config/parameters.yml  comme les identifiants de votre base de données, etc.

### Utilisation des services :

Exemple :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

class AdvertController extends Controller

{

public function addAction(Request $request)

{

// On récupère le service

$antispam = $this->container->get('oc\_platform.antispam');

// Je pars du principe que $text contient le texte d'un message quelconque

$text = '...';

if ($antispam->isSpam($text)) {

throw new \Exception('Votre message a été détecté comme spam !');

}

// Ici le message n'est pas un spam

}

}

## Créer un service avec des arguments :

Pour lui passer des arguments on utilise sa configuration.

# src/OC/PlatformBundle/Resources/config/services.yml

services:

oc\_platform.antispam:

class: OC\PlatformBundle\Antispam\OCAntispam

arguments:

- "@mailer"

- %locale%

- 50

Les arguments peuvent être :

* Des valeurs normales en YAML (des booléens, des chaînes de caractères, des nombres, etc.) ;
* Des paramètres (définis dans leparameters.ymlpar exemple) : l'identifiant du paramètre est encadré de signes « **%** » :%nomDuParametre%;
* Des services : l'identifiant du service est précédé d'une arobase :@nomDuService.

Dans cet exemple, notre service utilise :

* @mailer: le service Mailer (pour envoyer des e-mails) ;
* %locale%: le paramètre locale (pour récupérer la langue, définit dans le fichierapp/config/parameters.yml) ;
* 50: et le nombre 50 (qu'importe son utilité !).

<?php

// src/OC/PlatformBundle/Antispam/OCAntispam.php

namespace OC\PlatformBundle\Antispam;

class OCAntispam

{

private $mailer;

private $locale;

private $minLength;

public function \_\_construct(\Swift\_Mailer $mailer, $locale, $minLength)

{

$this->mailer = $mailer;

$this->locale = $locale;

$this->minLength = (int) $minLength;

}

/\*\*

\* Vérifie si le texte est un spam ou non

\*

\* @param string $text

\* @return bool

\*/

public function isSpam($text)

{

return strlen($text) < $this->minLength;

}

}

isSpam est aussi modifié, maintenant on ne touche plus qu’à la config au lieu du code si on veut modifier des valeurs.

Comme on a fait avec Mailer, si on a besoin d’un service dans un service, on le passe en arguments dans les paramètres et Symfony se charge d’instancier la class.

# La couche métier : les entités :

On utilise un ORM : Object-Relation-manager qui se charge d’enregistrer les informations à notre place. Plus de phpmyadmin.

## Notion d’ORM :

Si on dispose d’une variable $variable contenant un objet User qui représente une personne qui s’inscrit.

On se contentera de faire :

$orm->save($utilisateur) ;

Il faut penser objet et non plus SQL.

Un objet à enregistrer s’appelle une **ENTITE** et on dit **PERSISTER** une entité et non enregistrer …

## Créer une première entité avec doctrine :

Ca ressemblerait à ça, avec son utilisation :

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

class Advert

{

protected $id;

protected $content;

// Et bien sûr les getters/setters :

public function setId($id)

{

$this->id = $id;

}

public function getId()

{

return $this->id;

}

public function setContent($content)

{

$this->content = $content;

}

public function getContent()

{

return $this->content;

}

}

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use OC\PlatformBundle\Entity\Advert;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

class AdvertController extends Controller

{

public function viewAction()

{

$advert = new *Advert*;

$advert->setContent("Recherche développeur Symfony3.");

return $this->render('OCPlatformBundle:Advert:view.html.twig', array(

'advert' => $advert

));

}

}

### Les commentaires :

On utilise des commentaires pour ajouter des fonctionnalités, ils s’appellent **ANNOTATIONS**

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

// On définit le namespace des annotations utilisées par Doctrine2

// En effet, il existe d'autres annotations, on le verra par la suite,

// qui utiliseront un autre namespace

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* @ORM\Entity

\*/

class Advert

{

/\*\*

\* @ORM\Column(name="id", type="integer")

\* @ORM\Id

\* @ORM\GeneratedValue(strategy="AUTO")

\*/

protected $id;

/\*\*

\* @ORM\Column(name="date", type="date")

\*/

protected $date;

/\*\*

\* @ORM\Column(name="title", type="string", length=255)

\*/

protected $title;

/\*\*

\* @ORM\Column(name="author", type="string", length=255)

\*/

protected $author;

/\*\*

\* @ORM\Column(name="content", type="text")

\*/

protected $content;

// Les getters

// Les setters

}

**Important à noter qu’on utilise /\*\* et non /\* !**

Ça s’appelle **MAPPER** l’objet. [Documentation DOctrine2](https://symfony.com/doc/master/doctrine.html)

### Générer une entité avec le générateur :

Pour générer on fait :

1. **Console : Php bin/console doctrine :generate :entity**
2. **On donne le sous sous la forme : OCPlatformBundle :Advert**
3. **Enter car on utilise les annotations (par défaut)**
4. **On entre le nom des champs**
5. **Leur type**
6. **Facultatif ?**
7. **Unique ?**
8. **On recommence pour les autres champs**
9. **Entrée quand on a fini**

Maintenant entity.php est créé avec les annotations déjà prêtes (le code est faisable à la main).

### Affiner l’entité avec de la logique métier :

Exemple :

Si on a une entité **COMMANDE** avec :

ants :

* ListeProduitsqui contient un tableau des produits de la commande ;
* AdresseLivraisonqui contient l'adresse où expédier la commande ;
* Datequi contient la date de la prise de la commande ;
* Etc.

Les attributs devront être **MAPPE** c’est-à-dire défini comme des colonnes pour l’ORM via des annotations.

Mais certains résultats demandent des calculs :

<?php

// Exemple :

class Commande

{

public function getPrixTotal()

{

$prix = 0;

foreach($this->getListeProduits() as $produit) {

$prix += $produit->getPrix();

}

return $prix;

}

}

Si on crée getListeProduits dans l’objet.

Nos entités étant des objets, on fait appel au constructeur et définir des variables :

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* Advert

\*

\* @ORM\Table()

\* @ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\AdvertRepository")

\*/

class Advert

{

// ...

public function \_\_construct()

{

// Par défaut, la date de l'annonce est la date d'aujourd'hui

$this->date = new \Datetime();

}

// ...

}

## Tout sur le mapping :

[Documentation sur le mapping](http://docs.doctrine-project.org/projects/doctrine-orm/en/latest/reference/basic-mapping.html).

### Entity :

S’applique à une classe, elle définit un objet comme étant une entité, elle sera donc persistée par ORM.

@ORM\Entity

Il existe un paramètre facultatif qui précise le namespace.

@ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\AdvertRepository")

### Class :

S’applique aussi à une classe. Facultatif il permet de préciser le nom :

@ORM\Table(name="oc\_advert")

### Column :

Définit les caractéristiques de la colonne concernée.

Plusieurs types :

| **Type Doctrine** | **Type SQL** | **Type PHP** | **Utilisation** |
| --- | --- | --- | --- |
| string | VARCHAR | string | Toutes les chaînes de caractères jusqu'à 255 caractères. |
| integer | INT | integer | Tous les nombres jusqu'à 2 147 483 647. |
| smallint | SMALLINT | integer | Tous les nombres jusqu'à 32 767. |
| bigint | BIGINT | string | Tous les nombres jusqu'à 9 223 372 036 854 775 807. Attention, PHP reçoit une chaîne de caractères, car il ne supporte pas un si grand nombre (suivant que vous êtes en 32 ou en 64 bits). |
| boolean | BOOLEAN | boolean | Les valeurs booléennestrueetfalse. |
| decimal | DECIMAL | double | Les nombres à virgule. |
| date ou datetime | DATETIME | objet DateTime | Toutes les dates et heures. |
| time | TIME | objet DateTime- | Toutes les heures. |
| text | CLOB | string | Les chaînes de caractères de plus de 255 caractères. |
| object | CLOB | Type de l'objet stocké | Stocke un objet PHP en utilisantserialize/unserialize. |
| array | CLOB | array | Stocke un tableau PHP en utilisantserialize/unserialize. |
| float | FLOAT | double | Tous les nombres à virgule. Attention, fonctionne uniquement sur les serveurs dont la locale utilise un point comme séparateur. |

@ORM\Column(type="string")

Plusieurs paramètres facultatifs :

| **Paramètre** | **Valeur par défaut** | **Utilisation** |
| --- | --- | --- |
| type | string | Définit le type de colonne comme nous venons de le voir. |
| name | Nom de l'attribut | Définit le nom de la colonne dans la table. Par défaut, le nom de la colonne est le nom de l'attribut de l'objet, ce qui convient parfaitement. Mais vous pouvez changer le nom de la colonne, par exemple si vous préférez « isExpired » en attribut, mais « is\_expired » dans la table. |
| length | 255 | Définit la longueur de la colonne. Applicable uniquement sur un type de colonnestring. |
| unique | false | Définit la colonne comme unique. Par exemple sur une colonne e-mail pour vos membres. |
| nullable | false | Permet à la colonne de contenir desNULL. |
| precision | 0 | Définit la précision d'un nombre à virgule, c'est-à-dire le nombre de chiffres en tout. Applicable uniquement sur un type de colonnedecimal. |
| scale | 0 | Définit le *scale* d'un nombre à virgule, c'est-à-dire le nombre de chiffres après la virgule. Applicable uniquement sur un type de colonnedecimal. |

Pour définir plusieurs options en même temps, il faut simplement les séparer avec une virgule. Par exemple, pour une colonne « e-mail » enstring255 et unique, il faudra faire :

@ORM\Column(type="string", length=255, unique=true)

# Manipuler ses entités avec doctrine :

## Matérialiser les tables en BDD :

Pour créer la table :

**Php bin/console doctrine :database :create**

C:\wamp\www\Symfony>php bin/console doctrine:database:create

Created database `symfony` for connection named default

Ensuite on génère les tables à l’intérieur :

php bin/console doctrine:schema:update --dump-sql

On fait le point entre ce qu’on a dans nos entités et l’état de la BDD. Ici on fait le point sur ce qui va être fait **MAIS RIEN N’EST FAIT !**

Il faut forcer l’action :

C:\wamp\www\Symfony>php bin/console doctrine:schema:update --force

Updating database schema...

Database schema updated successfully! "1" query were executed

Maintenant la table est mise à jour.

### Modifier une entité :

Pour modifier une entité il suffit de créer un attribut et de lui rattacher l’annotation correspondante (ligne 18 à 21).

<?php

// src/OC/PlatformBundle/Entity/Advert.php

namespace OC\PlatformBundle\Entity;

use Doctrine\ORM\Mapping as ORM;

/\*\*

\* Advert

\*

\* @ORM\Table()

\* @ORM\Entity(repositoryClass="OC\PlatformBundle\Entity\AdvertRepository")

\*/

class Advert

{

// ... les autres attributs

/\*\*

\* @ORM\Column(name="published", type="boolean")

\*/

private $published = true;

// ...

}

Ensuite, pour les getters et les setters, on peut les faire à la main ou utiliser le générateur :

C:\wamp\www\Symfony>php bin/console doctrine:generate:entities OCPlatformBundle:Advert

Generating entity "OC\PlatformBundle\Entity\Advert"

  > backing up Advert.php to Advert.php~

  > generating OC\PlatformBundle\Entity\Advert

**Au lieu d’utiliser entity qui génère tout une entité, on utilise entities.**

Ensuite on regarde si on a bien le SQL correspondant et on l’applique (2 commandes différentes).

## Enregistrer ses entités avec l’Entity Manager :

### Les services de Doctrine2 :

**Rappel :** un service est accessible dans tout le code.

<?php

$doctrine = $this->get('doctrine');

L﻿﻿a classe Controller de Symfony intègre un raccourci. Il fait exactement la même chose, mais est plus joli et permet l'autocomplétion :

<?php

$doctrine = $this->getDoctrine();

Il est possible d’utiliser plusieurs connexions BDD avec $doctrine->getConnection($name) ;

Il dispose aussi de :

**$doctrine->getManager($name) ;**

Qui récupère un ORM à partir de son nom.

### EntityManager :

<?php

$em = $this->getDoctrine()->getManager();

Mais sachez que, comme tout service qui se respecte, vous pouvez y accéder directement via :

<?php

$em = $this->get('doctrine.orm.entity\_manager');

L’EntityManager permet de dire à Doctrine de dire « pesiste moi ça ».

### Les repositories :

<?php

$em = $this->getDoctrine()->getManager();

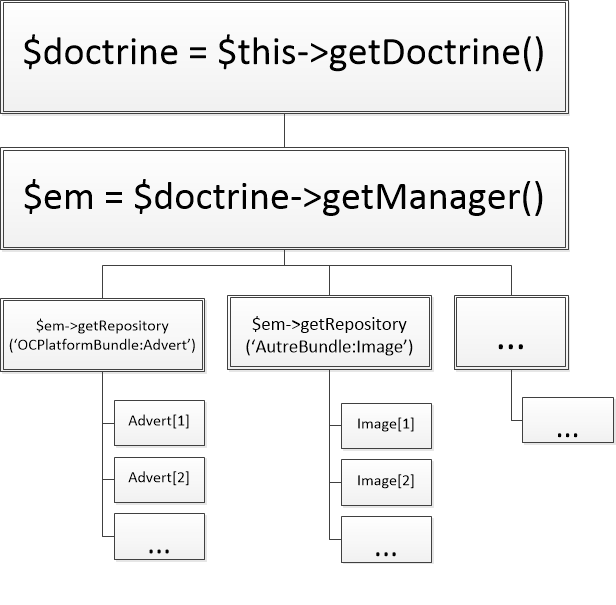
$advertRepository = $em->getRepository('OCPlatformBundle:Advert');

Il existe un repositorie par entité. Il faut toujours préciser son nom.

L’argument de la méthode getRepository est l'entité pour laquelle récupérer le repository. Il y a deux manières de spécifier l'entité voulue :

* Soit en utilisant le namespace complet de l'entité. Pour notre exemple, cela donnerait : 'OC\PlatformBundle\Entity\Advert'.
* Soit en utilisant le raccourci Nom\_du\_bundle:Nom\_de\_l'entité. Pour notre exemple, c'est donc 'OCPlatformBundle:Advert'. C'est un raccourci qui fonctionne partout dans Doctrine.

Attention, ce raccourci ne fonctionne que si vous avez mis vos entités dans le namespace Entitydans votre bundle.



### Enregistrer dans la base :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use OC\PlatformBundle\Entity\Advert;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpFoundation\Request;

class AdvertController extends Controller

{

public function addAction(Request $request)

{

// Création de l'entité

$advert = new *Advert*();

$advert->setTitle('Recherche développeur Symfony.');

$advert->setAuthor('Alexandre');

$advert->setContent("Nous recherchons un développeur Symfony débutant sur Lyon. Blabla…");

// On peut ne pas définir ni la date ni la publication,

// car ces attributs sont définis automatiquement dans le constructeur

// On récupère l'EntityManager

$em = $this->getDoctrine()->getManager();

// Étape 1 : On « persiste » l'entité

$em->persist($advert);

// Étape 2 : On « flush » tout ce qui a été persisté avant

$em->flush();

// Reste de la méthode qu'on avait déjà écrit

if ($request->isMethod('POST')) {

$request->getSession()->getFlashBag()->add('notice', 'Annonce bien enregistrée.');

// Puis on redirige vers la page de visualisation de cettte annonce

return $this->redirectToRoute('oc\_platform\_view', array('id' => $advert->getId()));

}

// Si on n'est pas en POST, alors on affiche le formulaire

return $this->render('OCPlatformBundle:Advert:add.html.twig');

}

}

* La ligne 15 permet de créer l'entité, et les lignes 16 à 18 de renseigner ses attributs ;
* La ligne 23 permet de récupérer l'EntityManager, on en a déjà parlé, je ne reviens pas dessus ;
* L'étape 1 dit à Doctrine de « persister » l'entité. Cela veut dire qu'à partir de maintenant cette entité (qui n'est qu'un simple objet !) est gérée par Doctrine. Cela n'exécute pas encore de requête SQL, ni rien d'autre.
* L'étape 2 dit à Doctrine d'exécuter effectivement les requêtes nécessaires pour sauvegarder les entités qu'on lui a dit de persister précédemment (il fait donc des INSERT INTO  & Cie) ;
* Ligne 36, notre Advert étant maintenant enregistré en base de données grâce au flush(), Doctrine2 lui a attribué un id ! On peut donc utiliser $advert->getId()  dans la génération de la route, et non un nombre fixe comme précédemment.

### Doctrine utilise les transactions :

On a fait $em->persist() et $em->flush() à cause des transactions. Si on fait un forum et qu’on enregistre un un sujet mais pas le message à cause d’une erreur … la BDD est cassée à cause des relations.

**Avec une transaction, on enregistre en même temps**. SI une échoue, les autres sont annulés.

Du coup : persist dit : garde en mémoir jusqu’au Flush, au Flush on enregistre.

### Doctrin simplifie la vie :

Persist() traite indifférament chaque requette : celle que l’on vient de créer comme celle que l’on a récupéré avec le repositorie. On ne se soucie pas des update et insert into.

<?php

// Depuis un contrôleur

$em = $this->getDoctrine()->getManager();

// On crée une nouvelle annonce

$advert1 = new *Advert*;

$advert1->setTitle('Recherche développeur.');

$advert1->setContent("Pour mission courte");

// Et on le persiste

$em->persist($advert1);

// On récupère l'annonce d'id 5. On n'a pas encore vu cette méthode find(),

// mais elle est simple à comprendre. Pas de panique, on la voit en détail

// dans un prochain chapitre dédié aux repositories

$advert2 = $em->getRepository('OCPlatformBundle:Advert')->find(5);

// On modifie cette annonce, en changeant la date à la date d'aujourd'hui

$advert2->setDate(new \Datetime());

// Ici, pas besoin de faire un persist() sur $advert2. En effet, comme on a

// récupéré cette annonce via Doctrine, il sait déjà qu'il doit gérer cette

// entité. Rappelez-vous, un persist ne sert qu'à donner la responsabilité

// de l'objet à Doctrine.

// Enfin, on applique les deux changements à la base de données :

// Un INSERT INTO pour ajouter $advert1

// Et un UPDATE pour mettre à jour la date de $advert2

$em->flush();

**Inutile de faire un persist(entity) quand on récupère une entité avec Doctrine. Quand on crée un objet, Doctrine n’est pas sensé savoir que c’est une entité que l’on va ajouter, alors que quand on récupère une entité, on sait déjà que c’est une entité.**

Il ne fera pas d’UPDATE non plus si on ne modifie rien.

### Les autres méthodes de l’EntityManager :

[Documentation officiel](http://www.doctrine-project.org/api/orm/2.5/class-Doctrine.ORM.EntityManager.html) pour toutes les voir.

**detach($entite)**  annule le persist()  effectué sur l'entité en argument. Au prochain flush() , aucun changement ne sera donc appliqué à l'entité. Voici un exemple :

<?php

$em->persist($advert);

$em->persist($comment);

$em->detach($advert);

$em->flush(); // Enregistre $comment mais pas $advert

**clear($nomEntite)**  annule tous les persist() effectués. Si le nom d'une entité est précisé (son namespace complet ou son raccourci), seuls les persist() sur des entités de ce type seront annulés. Si clear()  est appelé sans argument, cela revient à faire un detach()  sur toutes les entités d'un coup. Voici un exemple :

<?php

$em->persist($advert);

$em->persist($comment);

$em->clear();

$em->flush(); // N'exécutera rien, car les deux persists sont annulés par le clear

**contains($entite)** retourne true si l'entité donnée en argument est gérée par l'EntityManager (s'il y a eu un persist() sur l'entité donc). Voici un exemple :

<?php

$em->persist($advert);

var\_dump($em->contains($advert)); // Affiche true

var\_dump($em->contains($comment)); // Affiche false

**refresh($entite)** met à jour l'entité donnée en argument dans l'état où elle est en base de données. Cela écrase et donc annule tous les changements qu'il a pu y avoir sur l'entité concernée. Voici un exemple :

<?php

$advert->setTitle('Un nouveau titre');

$em->refresh($advert);

var\_dump($advert->getTitle()); // Affiche « Un ancien titre »

**remove($entite)** supprime l'entité donnée en argument de la base de données. Effectif au prochain flush(). Voici un exemple :

<?php

$em->remove($advert);

$em->flush(); // Exécute un DELETE sur $advert

## Récupérer ses entités avec un repositories :

D’abords on récupère le repositorie :

<?php

// Depuis un contrôleur

$repository = $this->getDoctrine()

->getManager()

->getRepository('OCPlatformBundle:Advert')

;

Puis,﻿﻿ depuis ce repository, il faut utiliser la méthode find($id) qui retourne l'entité correspondant à l'id $id. Exemple :

<?php

// src/OC/PlatformBundle/Controller/AdvertController.php

namespace OC\PlatformBundle\Controller;

use Symfony\Bundle\FrameworkBundle\Controller\Controller;

use Symfony\Component\HttpKernel\Exception\NotFoundHttpException;

class AdvertController extends Controller

{

public function viewAction($id)

{

// On récupère le repository

$repository = $this->getDoctrine()

->getManager()

->getRepository('OCPlatformBundle:Advert')

;

// On récupère l'entité correspondante à l'id $id

$advert = $repository->find($id);

// $advert est donc une instance de OC\PlatformBundle\Entity\Advert

// ou null si l'id $id n'existe pas, d'où ce if :

if (null === $advert) {

throw new *NotFoundHttpException*("L'annonce d'id ".$id." n'existe pas.");

}

// Le render ne change pas, on passait avant un tableau, maintenant un objet

return $this->render('OCPlatformBundle:Advert:view.html.twig', array(

'advert' => $advert

));

}

}

Il existe une autre syntaxe pour faire la même chose directement depuis l'EntityManager. Il s'agit de la méthode find de l'EntityManager, et non du Repository :

<?php

// Depuis un contrôleur

$advert = $this->getDoctrine()

->getManager()

->find('OCPlatformBundle:Advert', $id)

;