

**Aide-mémoire Symfony 2**

**Les bases résumées pour une utilisation optimale**

**KHAYAT Mehdi**

Sommaire

[I : Description de Symfony 2 2](#_Toc445912367)

[2 : Installation de Symfony 2](#_Toc445912368)

[3 : Les Bundles 3](#_Toc445912369)

[4 : Les composants clés de Symfony2 4](#_Toc445912370)

[a) Le routeur : 4](#_Toc445912371)

[Base d’une route 4](#_Toc445912372)

[Ajouter des paramètres à capturer dans l'URL : 4](#_Toc445912373)

[Ajouter des contraintes des paramètres de l'URL : 5](#_Toc445912374)

[Ajouter des paramètres facultatifs : 5](#_Toc445912375)

[Simplifier l’écriture de nos routes : 5](#_Toc445912376)

[b) Les Contrôleurs 6](#_Toc445912377)

[Définition : 6](#_Toc445912378)

[Manipuler les paramètres de la requête avec l’objet Request 7](#_Toc445912379)

[Construire une réponse efficace avec l’objet Response 9](#_Toc445912380)

[Manipuler la session 10](#_Toc445912381)

[Les Messages Flash 11](#_Toc445912382)

[c) Le Moteur de Templates Twig 12](#_Toc445912383)

[Définition : 12](#_Toc445912384)

[La syntaxe : 12](#_Toc445912385)

[Afficher des variables : 12](#_Toc445912386)

[Les Structures de contrôle et les expressions 15](#_Toc445912387)

[L’héritage et l’inclusion des templates : 16](#_Toc445912388)

# I : Description de Symfony 2

Symfony 2 est un framework (ou cadre de travail) PHP permettant d'utiliser du code propre et standardisé, ainsi que de faciliter le développement en ayant des briques logicielles préconstruites.

Les pours :

- Une productivité accrue (pas la peine de réécrire à chaque fois chaque élément, vu qu'ils sont déjà existant).

- Optimiser et maintenir plus facilement son site web, en effet le Framework étant standardisé, d'autres personnes peuvent facilement comprendre la structure et effectuer les modifications nécessaires.

- Une communauté très active, permettant de trouver de l'aide assez facilement

Les contres :

- Une courbe d'apprentissage plus longue, vu qu'il faut apprendre à utiliser chaque composant et apprendre les spécificités du framework.

- Avoir des connaissances en PHP, dans l'architecture MVC (au moins bien saisir le concept), avoir des notions dans la programmation orienté objet (POO) et dans les namespaces (Espaces de noms).

# 2 : Installation de Symfony

On vérifie que PHP 5 est bien installé sur le PC (ce qui est le cas avec wamp normalement)

Pour cela on utilise l'invite de commandes puis la commande : php -v

Si elle n'affiche pas la version de PHP, il faut ajouter le dossier de PHP dans les variables d'environnement de Windows. Pour cela on va dans :

Panneau de configuration -> Systèmes et sécurité -> Systèmes -> Paramètres systèmes avancés -> Variables d'environnement.

Dans le menu chercher dans la liste "PATH" puis indiquer le dossier ou est situé PHP. (en général il s'agira de le trouver dans wamp/bin/php).

Pour linux PHP est la plupart du temps installé de base (Paquet php5-cli)

Une fois ceci vérifié il est temps d'installer Symfony à proprement parler. Sur le site officiel, on télécharge le fichier nommé symfony.phar à [cette adresse](http://symfony.com/installer) puis on le décompresse dans le dossier web habituel (donc wamp/www/).

Ensuite se rendre dans le dossier correspondant puis clic droit suivi de "ouvrir une fenêtre de commande ici") (on peut aussi ouvrir une invite de commande et voyager jusqu'au dossier voulu.)

puis executer la commande suivante : php symfony.phar new Symfony

Attention sous linux à bien régler les droits sur les fichiers app/cache et app/logs sur lecture et écriture (chmod 777)

Et voila ! Symfony est désormais installé.

Petit détail avant de poursuivre, si le CSS de la page d'erreur est non visible, executez la commande : php app(ou bin)/console assets:install

# 3 : Les Bundles

Un bundle est une partie du site permettant de réaliser une tâche en particulier. En théorie il est possible de coder l'entièreté de son site dans un bundle, mais il est beaucoup plus pratique d'utiliser plusieurs bundles afin de rendre les fonctions plus facilement maintenables et interchangeables.

Pour cela, on commence par executer la commande : php app(ou bin)/console generate:bundle

Une fois ceci fait l'assistant d'installation du bundle démarre.

1 ère étape : le namespace, Usuellement, on l'écrit avec le nom de l'entreprise suivi du nom du bundle et le suffixe "Bundle" qui est obligatoire.

Par exemple : Adenis/SiteinterneBundle

2 ème étape : le nom du Bundle, ici rien à faire de particulier le nom proposé suffit amplement.

3 ème étape : la destination du Bundle, là non plus rien de particulier à faire car les bundles sont usuellement stockés dans le dossier "src".

4ème étape : Le format de configuration, sera pratique pour le fichier de configuration, par défaut on peut prendre le YAML (mais si vous connaissez bien les autres langages libre à vous).

Et voilà le bundle est désormais créé.

# 4 : Les composants clés de Symfony2

## a) Le routeur :

Le routeur, ou fichier de correspondances, permet d'indiquer où trouver chaque page ou Template du site web, (on peut le voir comme un panneau de signalisation géant). Et ce grâce à l'url de la barre d'adresses.

### Base d’une route

Adenis\_siteinterne\_home :

path : /accueil

defaults : {\_controller: AdenissiteinterneBundle:home:index}

Pour créer une route on a besoin des éléments suivants :

le nom de la route : arbitraire pour le nommage, mais indiqué par convention avec le nom du bundle racine, le nom du bundle courant puis l'action réalisée par le controlleur (ce nom peut être totalement arbitraire, mais il est pratique de bien se retrouver car ces noms sont réutilisés très souvent)

Path : il s'agit de l'URL qui sera accessible

Defaults : il s'agit ici des paramètres de la route (on a besoin au minimum du contrôleur et de l'action associée)

Attention à l'indentation en YAML qui se fait UNIQUEMENT avec 4 espaces !

Voici la route la plus basique possible, mais il est tout à fait possible de l'enrichir d'autres contraintes :

### Ajouter des paramètres à capturer dans l'URL :

Ajouter dans "path" les valeurs que l'on souhaite capturer

exemple : path : /accueil/{id}

A noter que l'on peut y ajouter plusieurs paramètres séparés par un slash ou un point.

Exemple : path : /accueil/{id]/{nom}.{date}

### Ajouter des contraintes des paramètres de l'URL :

Pour cela on utilise le mot clé "requirements" après defaults :

Exemple :

requirements :

id : \d{2}

date :\d{4}

(\d indiquant que la variable se doit d'être un nombre,

\d\* = tout nombre sauf 0

\d+ = tout nombre positif)

### Ajouter des paramètres facultatifs :

Pour cela, il suffit d'assigner une valeur par défaut pour les variables voulues, ce qui donne :

defaults :

\_controller : AdenissiteinterneBundle:home:index

(paramètre facultatif) : (valeur par défaut)

### Simplifier l’écriture de nos routes :

Pour simplifier l'écriture de nos routes, on peut y ajouter un préfixe. Pour cela on va dans le fichier routing.yml situé dans app/config et on y ajoute les lignes suivantes :

Adenis\_siteinterne:

resource :"@AdenissiteinterneBundle/resources/config/routing.yml (cela correspond au chemin de notre fichier de routage en passant).

prefix : /platform (ou tout autre préfixe voulu).

(à indiquer dans le contrôleur : générer des URL :

pour générer une URL on utilise la fonction suivante :

$url = $this->get('router')->generate('nom de la route',array('paramètres éventuels' => valeur de ceux ci));

mais il existe aussi une manière raccourcie :

$url = $this->generateUrl('nom de la route");

(à indiquer dans les vues : générer des URL :

Pour générer une URL depuis une vue on utilise la fonction "path" :

<a href = "{{path('Adenis\_siteinterne\_home'}}">

Lien vers la page d'accueil :D

</a> )

## b) Les Contrôleurs

### Définition :

Un contrôleur est un élément qui contient la logique de notre site internet. Concrètement, il utilise des services, les modèles, et il appelle la vue correspondante.

On peut dire qu'il s'agit du "chef d'orchestre" du système.

Voici ci-dessous un contrôleur basique :

<?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 !");

}

}

En gris : (en commentaire) la source du fichier

En Orange : Le namespace, il s'agit de créer un dossier de noms (afin de ne pas créer d'interférences si on apelle des fonctions de même nom dans le programme)

En violet : L'instanciation des objets utiles au contrôleur

En vert : le nom du contrôleur

En rouge : Permet d'appeler la fonction dont le nom est spécifié dans le routeur

En bleu : L'envoi de la réponse suite à l'appel de "indexAction();

Petite note ici : chaque fonction du contrôleur DOIT se terminer par "xxx**Action**". (pour les différencier des éventuelles fonctions/méthodes que l'on peut créer dans le contrôleur)

### Manipuler les paramètres de la requête avec l’objet Request

#### Les paramètres dans les routes :

Ici rien de plus simple, il suffit d’ajouter le nom de la variable du routeur dans la fonction correspondante.

Exemple :

Dans le routeur :

oc\_platform\_view:

path: /advert/{**id**}

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

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

{

// …

public function viewAction($id)

{

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

}

}

#### Les paramètres hors routes :

Dans le cas des paramètres non présents dans le routeur (comme dans l’url, les variables de formulaire etc…) on utilise l’objet « Request » dans la fonction, suivi de la méthode correspondante à ce que l’on cherche à récupérer.

Exemple :

Sur une adresse de ce type : /platform/advert/9?tag=développeur, on cherche à extraire le contenu de la variable « tag ».

<?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');

return new Response(

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

);

}

}

Note importante : Pour rechercher d’autres types de variables, on utilise la même méthode mais en changeant le « query » par un autre mot clé correspondant à ce que l’on veut extraire. Les mots clés utilisables sont indiqués ci-dessous :

| **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 | Inexistante car unique a symfony | On utilise $id dans les arguments de la méthode, mais vous pourriez également faire $request->attributes->get('id') |

### Construire une réponse efficace avec l’objet Response

L’objet Response est utile pour retourner… une réponse ! (oui je sais c’est tellement logique °°), Cependant il est important de savoir comment elle est construite et comment fournir une réponse adaptée à ce que l’on veut faire.

Basiquement, la méthode pour renvoyer une réponse est la suivante :

// 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;

Fort heureusement, on n’utilisera jamais cette méthode vu sa longueur et son inutile complexité.

Pour respecter l’architecture MVC, la réponse se trouvera simplement dans une vue, et non directement dans le contrôleur.

#### Appel d’une vue

Pour appeler la vue donc, on utilise cette méthode :

return $this->

get('templating')->

renderResponse(

'OCPlatformBundle:Advert:view.html.twig',array('id' => $id));

A noter qu’il existe une méthode encore plus raccourcie qui est la suivante :

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

Ceci nous permet donc d’afficher la vue appelée. (A noter qu’il suffit de remplacer ce qui est en rouge par le chemin de la vue pour l’obtenir)

Etant la plus courte et la plus simple à utiliser, on retiendra surtout la méthode render().

#### Exécuter une redirection :

Dans un grand nombre de cas nous devrons faire une redirection (Rediriger vers une autre page ou une autre route)

Pour cela on utilise la méthode suivante :

return $this->redirectToRoute('oc\_platform\_home'); oc\_platform\_home étant le nom de la route.

### Manipuler la session

Ce qu’il est aussi possible de faire avec un contrôleur, c’est de manipuler les variables relatives à la session utilisateur. Ceci peut être utile pour fournir des pages différentes selon les utilisateurs par exemple.

Voici ci-dessous les méthodes pour manipuler ces variables :

* Récupérer la session :

$session = $request->getSession();

* Récupérer le contenu de la variable utilisateur (user\_id)

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

* Définition d’une nouvelle valeur pour la variable utilisateur

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

### Les Messages Flash

Les messages Flash sont des messages avec la particularité de n’apparaître que sur une seule et unique page.

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));

}

A noter que cette fois le template doit contenir une zone pour afficher le message flash voulu :

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

<!DOCTYPE html>

<html>

<head>

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

</head>

<body>

<h1>Affichage d'une annonce</h1>

<p>

{# 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 %}

</p>

<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>

On peut notamment remarquer que ce qui est en rouge n’est que le chemin contenant l’information recherché. Il suffit donc de l’assigner à une variable twig (ici message) pour la faire afficher.

## c) Le Moteur de Templates Twig

### Définition :

Twig (brindille en français) est un pseudo langage permettant de simplifier et surtout de séparer le code PHP du code HTML. De plus, il permet aussi de facilement générer du code dynamique sans passer par du PHP brut.

Ce pseudo langage s’utilise dans des templates Twig, qui sont simplement des fichiers HTML portant l’extension. « . twig » à la fin.

L’autre intérêt de Twig est qu’il gère automatiquement la sécurité des données transmises.

Enfin, l’utilisation de Twig ne se limite pas qu’aux pages HTML mais, peut aussi être utilisé pour des mails par exemple. Bref, tout ce qui concerne les vues peut être géré par TWIG !

### La syntaxe :

Voici ci-dessous les différents types de syntaxes trouvables dans Twig :

* {{…}} : Sert à afficher quelque chose
* {%...%} : Sert à effectuer quelque chose
* {#...#} : Sert à commenter quelque chose

### Afficher des variables :

On se rappelle : Pour afficher des variables, on se sert des doubles accolades {{…}}, suivi du nom de la variable ainsi que différents paramètres selon ce que l’on a besoin de faire. Voici un tableau résumant le tout :

| **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; ?> |

On remarque dans ce tableau que chaque méthode twig a son équivalent PHP, mais aussi que les filtres sont séparés par des « | » entre chacun d’entre eux.

#### Les Filtres

En parlant de filtres, en voici quelques-uns utiles :

| **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 deDatetime. | {{ 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](http://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 }} |

Cette liste n’est évidemment pas exhaustive, car il est tout à fait possible de créer nos propres filtres

#### Les variables globales

Il est possible de récupérer certaines variables directement depuis le template grâce au la variable globale {{app}} présente dans twig, il suffit ensuite d’appliquer le bon attribut pour extraire les variables désirées).

| **Variable** | **Description** |
| --- | --- |
| {{ app.request }} | La requête « request » qu'on a vu 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. |

D’ailleurs comme pour les filtres, il est possible de définir ses propres variables globales.

### Les Structures de contrôle et les expressions

Après avoir appris comment afficher quelque chose, il est temps de comprendre comment exécuter quelque chose, et on y parvient à l’aide des structures de contrôle.

La condition :

Elle se note :

{% if (variable) (signe < ou > ou !) (Autre variable) %}

Condition 1

{%else%}

Condition 2

{%endif%}

Elle permet de tester une variable et selon qu’elle indique permet différentes réponses.

Les tests utiles avec if :

Defined : (l’équivalent de isset en PHP Brut)

{% if (variable) is defined %} … {% endif %}

La boucle :

Elle se note :

{% for (valeur) in (tableau) %}

Instructions

{%endfor%}

Définition :

Elle se note :

{% set (nom de la variable) = (valeur de la variable) %}

Et permet de euh…. Bah de définir une valeur à une variable °°.

### L’héritage et l’inclusion des templates :

#### L’héritage de template

Comment créer un design efficace sans pour autant réécrire chaque page ? En utilisant les Template et leur système d’héritage ! On remarque d’ailleurs que le PHP « brut » permet aussi de le faire grâce aux include(). En soi c’est très pratique de procéder ainsi car cela nous permet de découpler les éléments de la page, permettant de travailler facilement dessus sans problèmes et sans interférences.

Le principe est le suivant :

* On intègre un Template « père » qui détient le design général du site, avec des trous à l’endroit où les éléments de page seront insérés. On appelle ces trous « blocks ».
* Pour combler les trous réalisés précédemment, on crée des Template « fils » incluant la Template père et indiquant quels « block » doivent être remplis.

Remarque : Quand une Template n’est utilisée en lien avec aucun contrôleur particulier, on doit le mettre dans le dossier « racine » des vues, de sorte à ce qu’il soit accessible facilement par tous les autres Templates fils (c’est juste plus commode de suivre cette convention pour simplifier l’écriture des chemins d’accès des Templates).

En pratique :

Voici les Templates suivants :

Template père :

{# 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 #}

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

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

{% block body %}

Ce block contiendra notre body et comblera le trou laissé dans le template père.

{% endblock %}

Décryptons un peu ce code :

En vert on a défini un block qui servira pour le titre, il apparaitra à chaque page du site pour peu que l’on fasse appel à lui. Comment faire appel à lui ? en utilisant la fonction {{ parent() }} de cette façon il s’affichera, même si d’autres éléments sont indiqués à côté de lui, comme l’index par exemple.

En Bleu : On a défini un « trou » dans la page en utilisant la méthode {%block (nom\_du\_block) %} {%endblock%} dans la Template père. Pour le remplir par la suite, on utilise cette même méthode dans les templates fils mais cette fois en remplissant avec le contenu désiré.

En rouge : Très important ! Il permet d’indiquer que le template fils à un père, et donc qu’il hérite de l’ensemble de ses propriétés, c’est en somme un « lien de parenté » entre le père et son fils (c’est beau tout cet amour <3)… On le construit en indiquant simplement la balise {%extends « chemin de la template père »%}

Grâce à cela on sait désormais comment créer et compléter des templates en découplant leurs éléments. Cependant, une technique a vu le jour permettant de découper efficacement les éléments d’une page. On la nomme « Le Triple héritage ».

Le triple héritage :

C’est très simple, le triple héritage consiste à répartir l’ensemble des éléments d’une page sur trois niveaux :

* Le layout général : Ce qui contiendra tout le design général du site, et qui sera réutilisé dans chacune de ses pages. (syntaxe : « ::layout.html.twig »)
* Le layout du bundle : Ici on se spécialise un peu plus en ajoutant tout ce qui a trait aux bundles utilisables dans la page. (syntaxe : « nom\_bundle ::layout.html.twig)
* La template de page : gère la page courante, donc sera différente entre chaque page. (syntaxe : « nom\_bundle :nom\_contrôleur :nom\_template)

A noter que le layout général sera créé et ajouté dans le répertoire app/Resources/view

#### L’inclusion de templates

Quand faire de l’inclusion ?

Lorsqu’un même élément va se retrouver sur plusieurs pages, il est intéressant de faire une inclusion (c’est le cas des formulaires par exemple). On remarquera que l’élément inclus dans les pages n’a généralement pas de Template héritée.

Pour simplifier : plusieurs pages peuvent hériter d’une ou plusieures template, par contre une template incluse dans une autre ne pourra hériter de personne.

(Peut être qu’un schéma serait intéressant à placer ici).

En pratique :

On utilise la fonction {{ include (« nom\_bundle :nom\_contrôleur :nom\_de\_la\_template\_à\_inclure »)}}

En résumé, l’include consiste en un simple copier-coller d’un morceau de code.

#### L’inclusion des contrôleurs

Après l’inclusion des templates, voici l’inclusion des…. CONTROLEURS !!!

Pourquoi inclure des contrôleurs ?

L’intérêt est que l’on n’a pas toujours les variables directement dans la template, mais il se peut qu’elle se trouve dans la base de données. Ducoup, l’include simple de templates ne suffit donc plus (logique car elles ne sont pas en dur dans la template incluse). Pour remédier à cela, on inclut à la place une action du contrôleur.

Un exemple pour illustrer le principe consiste par exemple dans une liste de petites annonces. Ces données ne seront évidemment pas écrit dans le code de la template, mais se trouvera dans la base de données. Or l’include duplique juste le code de la template, sans se soucier de la base de données.

Ce que le contrôleur peut faire, il suffit de programmer une action dans le contrôleur pour accéder à la base de données, puis passe ces valeurs à la template qui gère les annonces. Puis celle-ci sera finalement incluse dans le layout ou dans toute autre template qui aura besoin d’elle.

Enfin, pour inclure une action d’un contrôleur, on utilise la méthode suivante :

{{ render(controller(« nom\_Bundle :nom\_contôleur :nom\_action ») }}

Ceci conclut le chapitre sur le moteur de templates TWIG.