Le Framework PHP Symfony

8/02/2016

Thomas BIZOUERNE
Architecte
sopra steria

Sommaire

- Introduction
- Historique de Symfony
- Concepts de base – pré-requis
- Présentation de Symfony
- Symfony en pratique
- Symfony – notions avancées
- Autres outils du framework
- Bibliographie
- Conclusion

Introduction

Introduction

- **Thomas Bizouerne**
  - Sorti Major de la MIAGE d'Orléans en 2008
  - Architecte chez Sopra Steria

- Contexte
  - Le langage PHP
  - Développement d'applications web

Introduction

- Développement classique en PHP

```
<?php
$link = mysql_connect('localhost', 'myuser', 'mypassword');
mysql_select_db('blog_db', $link);
$result = mysql_query('SELECT id, title FROM post', $link);
?>
<!DOCTYPE html>
<html>
    <head>
        <title>List of Posts</title>
    </head>
    <body>
        <h1>List of Posts</h1>
        <ul>
            <?php while ($row = mysql_fetch_assoc($result)): ?>
            <li>
                <a href="/show.php?id=<?php echo $row['id'] ?>">
                    <?php echo $row['title'] ?>
                </a>
            </li>
            <?php endwhile; ?>
        </ul>
    </body>
</html>
<?php mysql_close($link); ?>
```

- Pas de gestion d'erreur
- Aucune séparation de code
- Réutilisation de code impossible

Symfony

- Lancement de **symfony-project.com** en **2005**
- **Framework Web MVC** en **PHP**
- Créé par **SensioLabs** (**Fabien Potencier**), agence Web française
- Utilise d'autres projets PHP : Doctrine, Propel, PDO, PHPUnit, Twig, …
- Utilisé par d'autres projets : Drupal, Dailymotion, Laravel, …
- Utilisé par de nombreux sites à travers le monde
- Philosophie : **ne pas réinventer la roue**
- **http://symfony.com/**

Historique

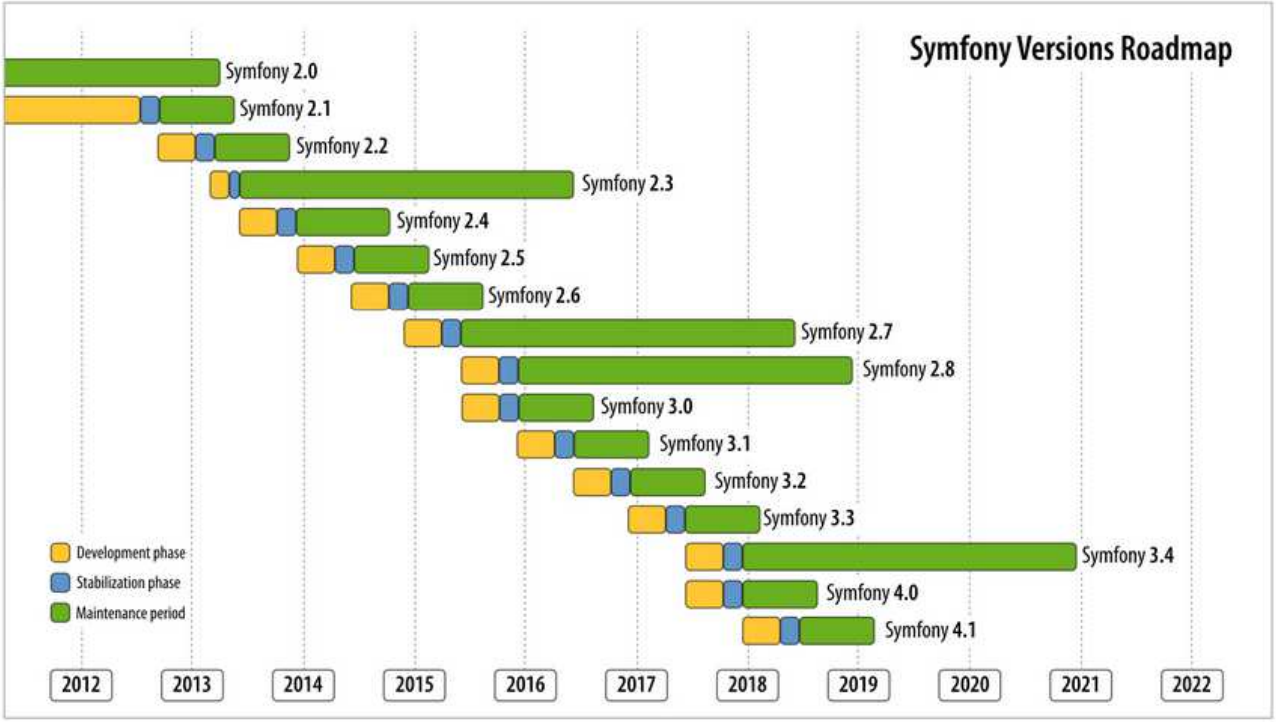
Historique

Version	Date de sortie	Version de PHP
1.0	Janvier 2007	>= 5.0
1.3	Novembre 2009	>= 5.2.4
1.4	Novembre 2009	>= 5.2.4
2.0	Juillet 2011	>= 5.3.2
2.1	Septembre 2012	>= 5.3.3
2.2	Mars 2013	>= 5.3.3
2.3 (LTS)	Mai 2013	>= 5.3.3
2.4	Novembre 2013	>= 5.3.3
2.5	Mai 2014	>= 5.3.3
2.6	Novembre 2014	>= 5.3.3
2.7 (LTS)	Mai 2015	>= 5.3.9
2.8 (LTS)	Novembre 2015	>= 5.3.9
3.0	Novembre 2015	>= 5.5.9

Historique

Version	Date de sortie	Version de PHP
3.1	Mai 2016	>= 5.5.9
3.2	Novembre 2016	>= 5.5.9
3.3	Mai 2017	>= 5.5.9 | >=7.0.8
3.4 (LTS)	Novembre 2017	>= 5.5.9 | >=7.0.8
4.0	Novembre 2017	7.1.3
4.1	Mai 2018	
4.2	Novembre 2018	
4.3	Mai 2019	
4.4 (LTS)	Novembre 2019	
5.0	Novembre 2019	
…	…	**LTS** : Long-Term Support

Historique



- **Yellow** represents the Development phase
- **Blue** represents the Stabilization phase
- **Green** represents the Maintenance period

Concepts de base – pré-requis

Concepts de base – pré-requis

- Qu'est ce qu'un Framework ?
- Qu'est ce que MVC ?
- Qu'est ce qu'un ORM ?

Framework

Qu'est ce qu'un Framework de développement ?

- Littéralementttt, le mot « **framework** » signifie « cadre de travail »
- Boite à outils de composants logiciels pour créer :
  - La structure d'une application
  - Les grandes lignes d'une application
  - Du code « bien organisé »
- L'apprentissage d'un framework est un investissement : il y a un certain effort à fournir au début, mais les résultats se récoltent ensuite sur le long terme !

Framework

Qu'est ce qu'un framework orienté objet ?

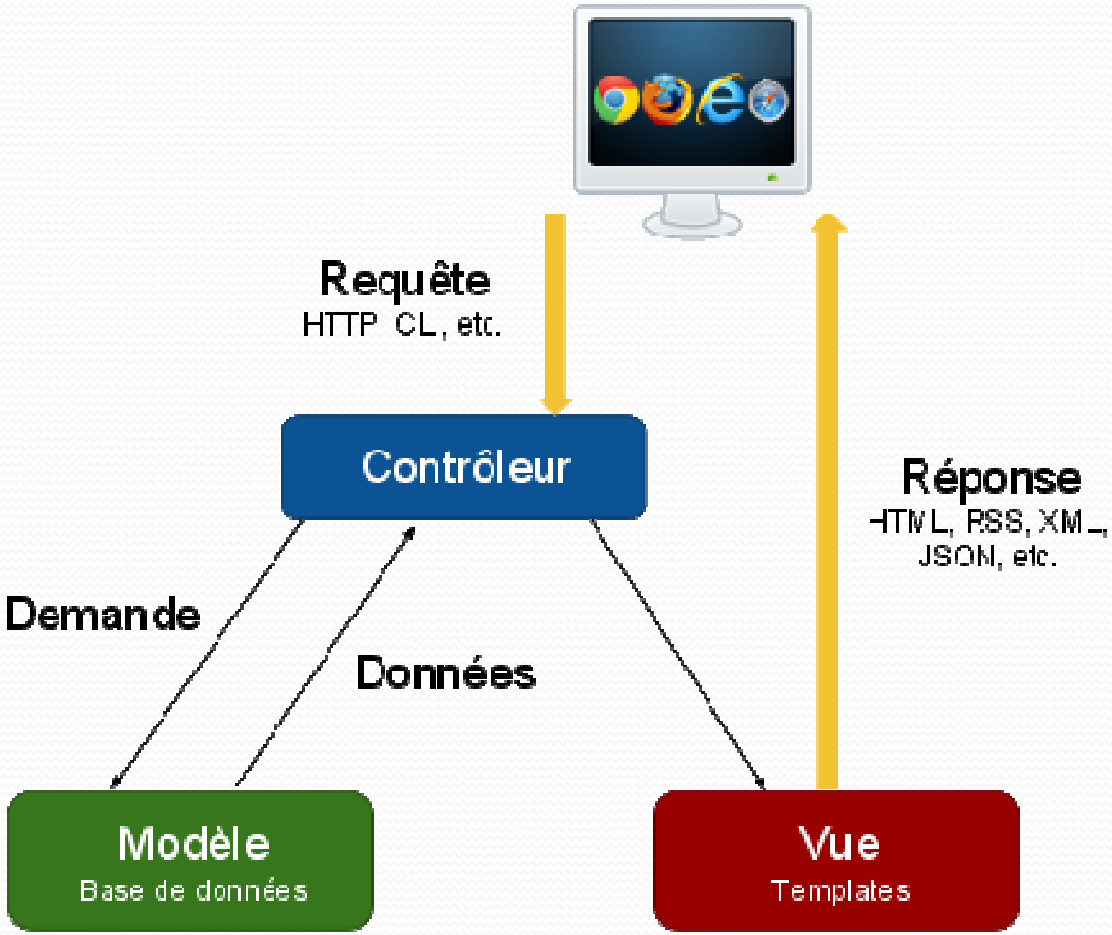
- En programmation orientée objet, un framework est typiquement composé de classes mères qui seront dérivées et étendues par héritage en fonction des besoins spécifiques à chaque logiciel qui utilise le framework.

MVC

Qu'est ce que MVC (Modèle – Vue – Contrôleur)

- Manière d'architecturer une application
- Séparation du code en différentes couches logiques

MVC : schéma


Requête
HTTP, CLI, etc.
Contrôleur
Réponse
HTML, RSS, XML,
JSON, etc.
Demande
Données
Modèle
Base de données
Vue
Templates

MVC : le modèle

Le modèle englobe

- Les données de l'application
- L'accès de différentes manières à ces données
- Les traitements sur les données de l'application

MVC : le modèle

Le modèle

Exemple avec une base de données

Le modèle englobe :

- La base de données
- Les méthodes pour récupérer les données
- Les méthodes pour mettre à jour les données

MVC : la vue

La Vue

- Son rôle est d'afficher les pages
- Assure la présentation des données du modèle
- N'effectue aucun traitement !

MVC : le contrôleur

Le Contrôleur (le chef d'orchestre)

- Son rôle est de générer la réponse à la requête HTTP demandée par un visiteur
- Sait réagir aux évènements de l'utilisateur
- Attribue les actions correspondantes
- Se contente la plus part du temps « d'utiliser » les deux autres composants : les modèles et les vues

ORM

Qu'est ce qu'un ORM (Object-Relational Mapper)

- Contexte
  - Une application qui manipule des objets
  - Une base de données relationnelle de stockage de données
- Problème
  - Comment faire le mapping ?
- Dans Symfony, l'ORM fourni par défaut est **Doctrine**
  - Doctrine est un « Data mapper » qui assure le lien entre les entités et la BDD via les annotations et les repositories.

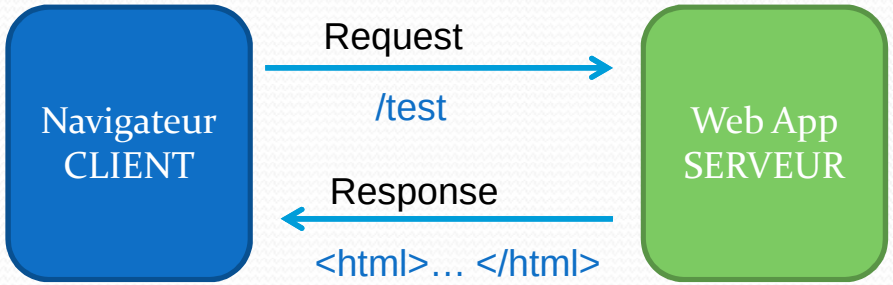
ORM

Qu'est ce qu'un ORM (Object-Relationnal Mapper)

- Accès aux données en BDD avec ORM
  - L'ORM convertit les manipulations d'objet métier en requêtes SQL
- Création, récupération d'objets avec l'ORM
  - L'ORM transforme les données venant de la BDD en objets métiers

Présentation de Symfony

Présentation de Symfony



- Request : demande de la part du client Web (navigateur)
  - POST, GET, etc.
- Symfony traite la demande, construit la réponse et la retourne
- Response
  - Contient un code (200 OK, 404 Erreur, etc.)
  - et un contenu (HTML, PDF, etc.)

Présentation de Symfony

- Le but final d'une application Web est toujours de comprendre chaque requête puis de créer et retourner un résultat approprié.

- Symfony fournit deux classes qui vous permettent d'interagir avec les requêtes et réponses HTTP de façon plus simple :
  - La classe « **Request** » est une simple représentation orientée objet d'un message d'une requête HTTP
  - La classe « **Response** » est une simple représentation PHP d'un message de réponse HTTP
  - Les classes « **Request** » et « **Response** » sont des éléments du composant appelé « **symfony/http-foundation** » qui peut être utilisé pour tout projet PHP

Pratique du MVC dans Symfony

- **Contrôleur frontal**
  - C'est l'entrée du site
  - Il est nécessaire de faire de la récriture d'URL pour avoir des adresses propres
  - Il existe deux contrôleurs frontaux dans symfony :
    - app.php
    - app_dev.php

Pratique du MVC dans Symfony

- **Le routeur**
  - C'est l'aiguilleur
  - Une fois la porte franchie, il oriente vers le bon contrôleur et la bonne action en fonction de l'URL
- **Le contrôleur**
  - Un contrôleur regroupe un ensemble de méthodes réalisant chacune une action
  - Le contrôleur a en charge la communication avec le modèle et la vue

Pratique du MVC dans Symfony

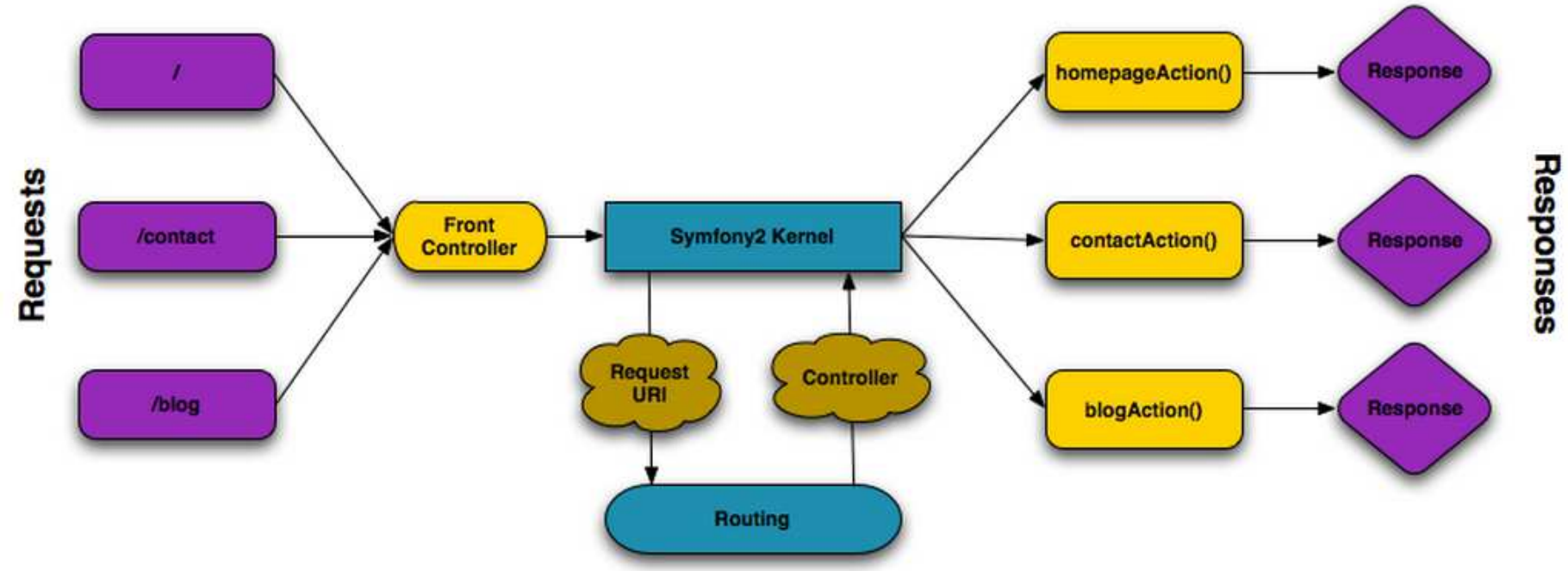
- **Le modèle**
  - Constitué des Entity et des Repository dans Symfony
  - Le modèle assure les accès à la BDD et peut contenir des traitements
- **La vue**
  - La vue, via un moteur de template (Twig en l'occurrence), affiche le contenu envoyé par le contrôleur

Deux composants essentiels dans Symfony

- **Le Kernel**
  - C'est le cœur ou noyau du framework
  - Il envoie des évènements qu'il est possible d'écouter
- **Le service container**
  - Il contient des services prédéfinis ou créés par le développeur

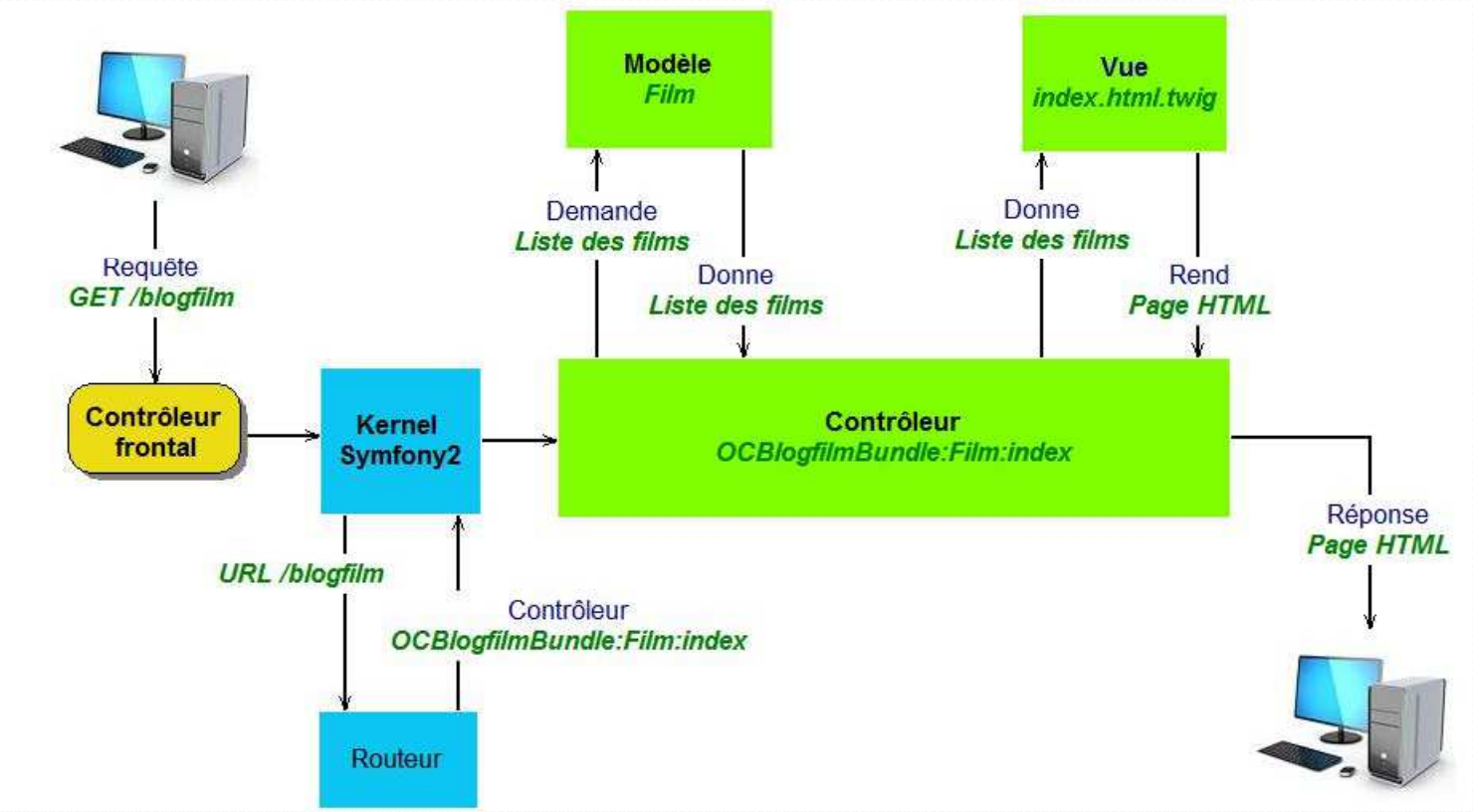
Présentation de Symfony

- Symfony structure ce mécanisme (requête ➔ réponse)



- Les requêtes entrantes sont interprétées par le routing et passées aux actions des contrôleurs qui retournent des objets Response

Parcours d'une requête dans Symfony



Présentation de Symfony

Symfony propose :

- Une séparation du code (MVC)
- Un templating simple (Twig)
- Des performances optimisées (système de cache)
- Du formatage d'URL
- Un système de configuration en cascade et multi-formats

Symfony en pratique

Symfony en pratique

Préparation d'un environnement de dev (Windows)

- Installer Wamp Server :
  - http://www.wampserver.com/
- Ajouter le dossier contenant « **php.exe** » à la variable d'environnement PATH
  - Exemple : **C:\wamp\bin\php\php5.6.25**
- Commande pour vérifier la version de php :
  - **php –v**
- Il existe au moins deux façons d'installer Symfony :
  - par l'utilisation de Composer
  - par l'installeur de Symfony

Symfony en pratique

Installation de Symfony avec Composer

- Installer Composer sous : **C:\wamp\www**
  - **php -r "eval('?>'.file_get_contents('http://getcomposer.org/installer'));"**
- Commande pour tester l'installation de Composer :
  - **php composer.phar --version**
- Commande de mise à jour de Composer (si besoin est) :
  - **php composer.phar self-update**
- Commande de création de projet Symfony sous : **C:\wamp\www**
  - **php composer.phar create-project symfony/framework-standard-edition OCBlogcinema "3.2.4"**

Symfony en pratique

Arborescence d'un projet Symfony

- Un projet Symfony se présente ainsi :

app/ →	Eléments communs à tous les bundles (configuration, cache, logs, layouts globaux)
bin/ →	Exécutables en rapport avec les commandes console
src/ →	Bundles du projet avec tous les éléments qui leurs sont propres (le code source)
var/ →	Logs, cache
tests →	Tests de l'application
vendor/ →	Librairies tierces (Symfony, Doctrine, Twig, etc.)
web/ →	Contrôleurs frontaux, CSS, JS…

Symfony en pratique

Application en ordre de marche

http://localhost/OCBlogcinema/web/app_dev.php

Welcome to

Symfony 3.2.4

Your application is now ready. You can start working on it at: `C:\wamp\www\OCBlogcinema\`

What's next?

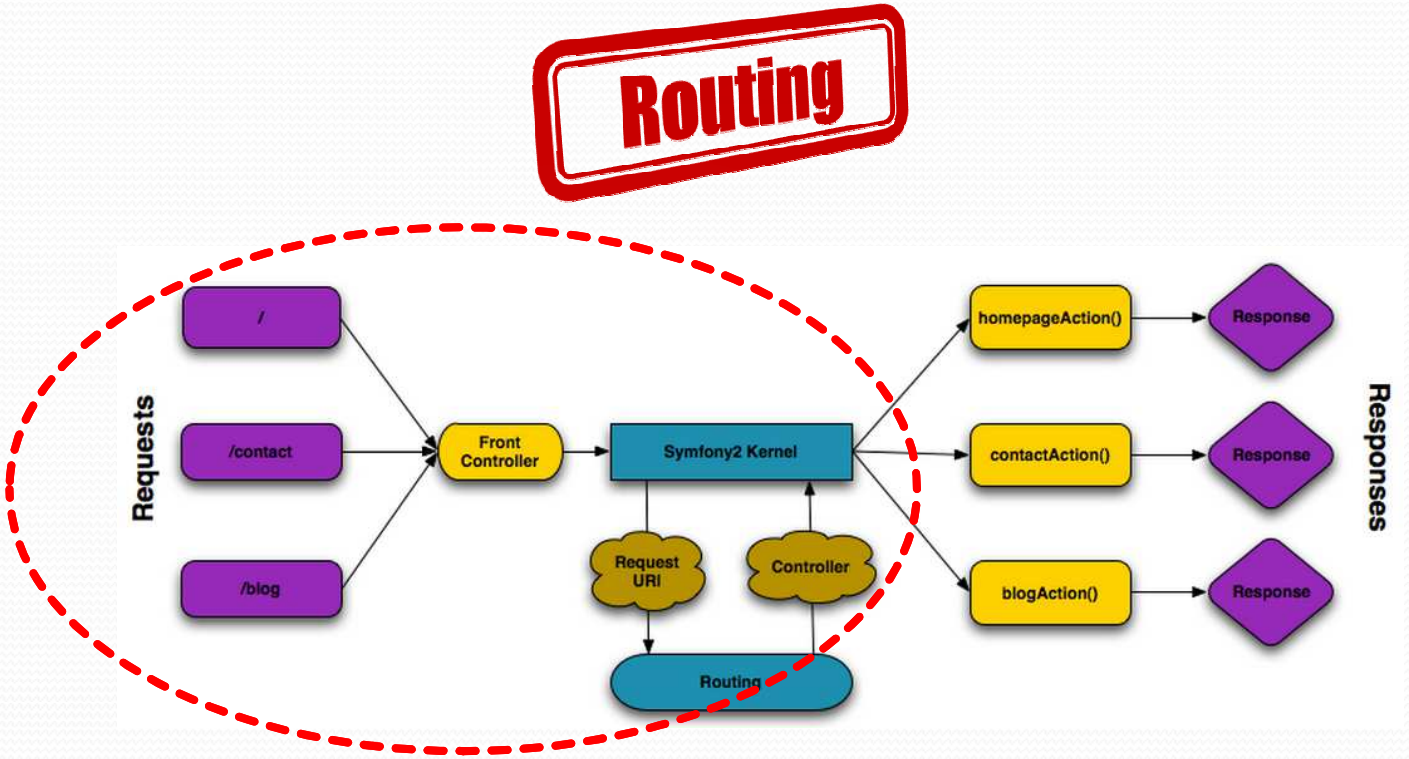
Read the documentation to learn

How to create your first page in Symfony

Symfony en pratique

Eléments d'un projet Symfony

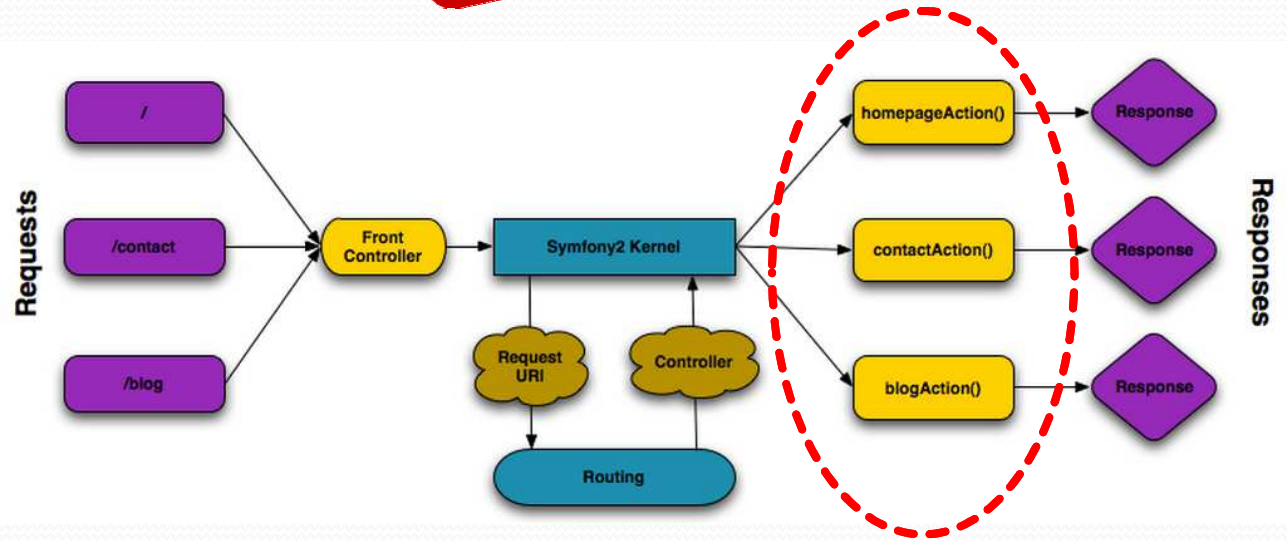
- Lorsque l'on écrit un projet avec Symfony, on écrit :
  - Une configuration faisant correspondre des URL à des Actions



Symfony en pratique

Eléments d'un projet Symfony

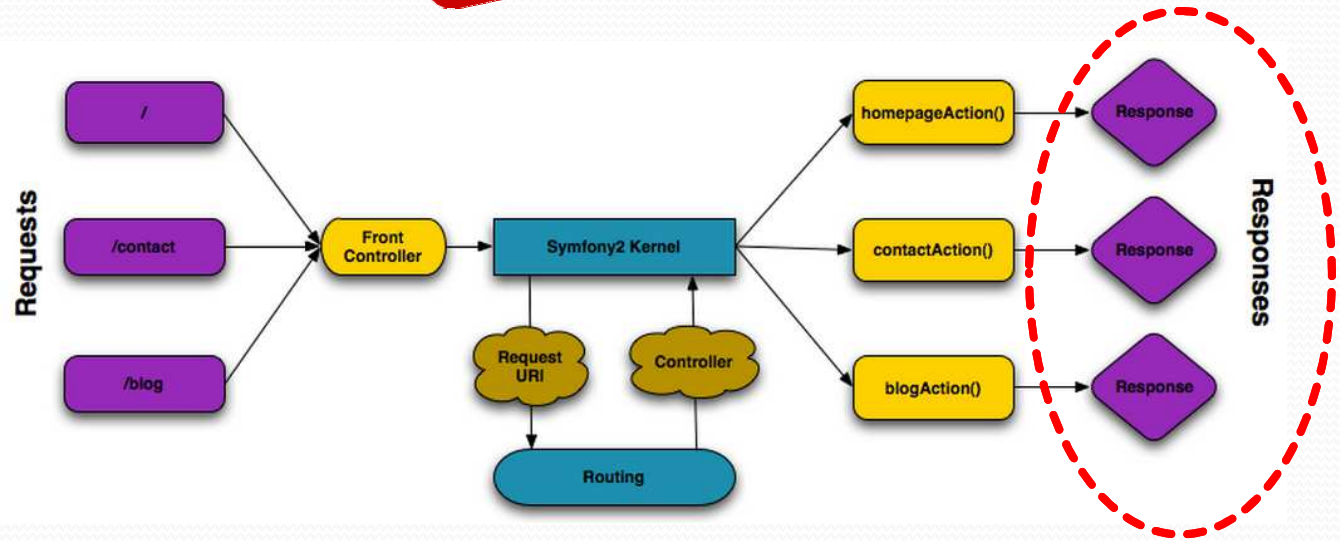
- Lorsque l'on écrit un projet avec Symfony, on écrit :
  - Des méthodes d'actions, qui font éventuellement appel à des données



Symfony en pratique

Eléments d'un projet Symfony

- Lorsque l'on écrit un projet avec Symfony, on écrit :
  - Des vues



Symfony en pratique

La console Symfony

- Console permettant d'exécuter certaines tâches souvent utiles lors du développement

- Simplifie grandement la vie du développeur

- Utilisation :
  - Se placer à la racine du projet
    - Exemple : **C:\wamp\www\OCBlogcinema**
  - Executer :
    - **php bin/console <commande>**

Symfony en pratique

Création d'un « bundle »

- Un bundle est un répertoire qui contient tout ce qui est relatif à une fonctionnalité de l'application (code PHP, configuration, template Twig, etc.)

- La commande de génération de bundle :

  php bin/console generate:bundle

  - Crée une arborescence type pour le bundle,
  - Déclare le bundle dans le fichier « noyau » qui charge tous les bundles nécessaires à l'application
  - Crée un fichier de configuration du routing pour la fonctionnalité

Namespace : structuration hiérarchique des classes, respecte généralement l'arborescence des répertoires.

Symfony en pratique

Structure du bundle

- Arborescence type créée par la commande de génération du bundle

src

- **OC**
  - **BlogBundle**
    - **Controller**
    - **DependencyInjection**
    - **Resources**
    - **Tests**

Symfony en pratique

Le routing

- La création du bundle a entraîné :
  - La création d'un fichier de configuration du routing propre au bundle
    - src/OC/BlogBundle /Resources/config/routing.yml
  - La déclaration de ce fichier dans la configuration générale (= import)
    - app/config/routing.yml

Symfony en pratique

Le routing en théorie

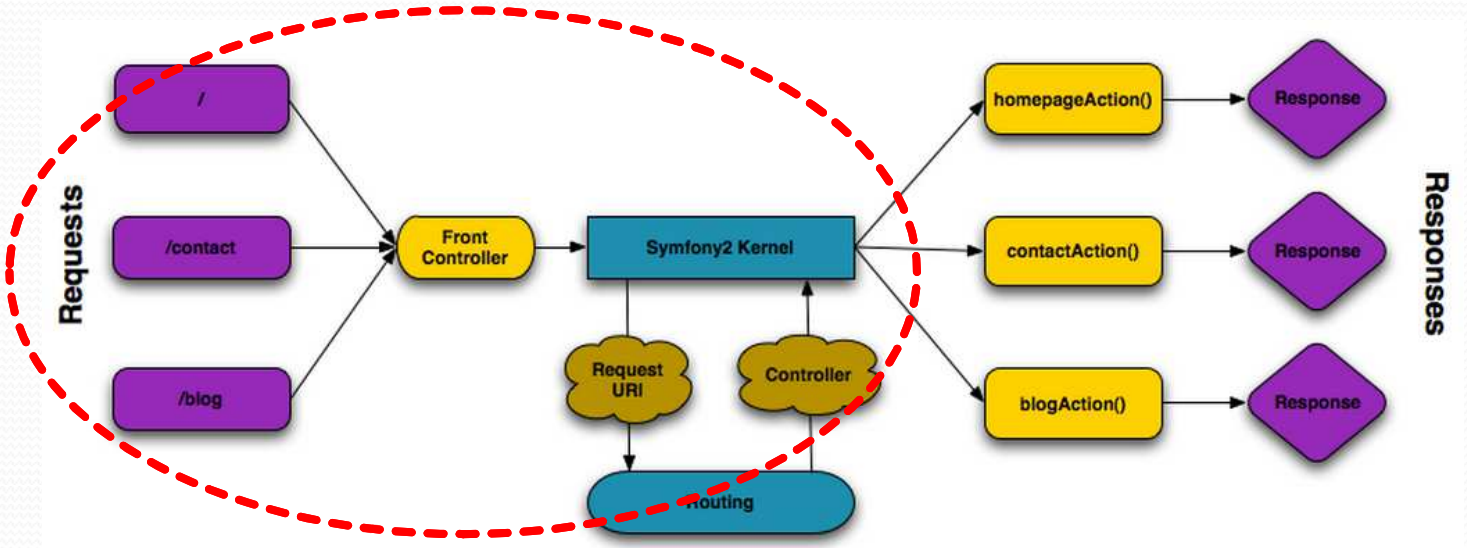
- Définition
  - Le routing est la manière de faire correspondre une URL à une ressource web.
- Les URLs
  - L'url est l'interface entre l'utilisateur et le site
  - Généralement, une url expose la structure du site
  - Dans la vision Symfony, ces urls doivent être parlantes
- Symfony possède donc un composant spécifique, dédié à la gestion des urls

Symfony en pratique

Le routing, concrètement

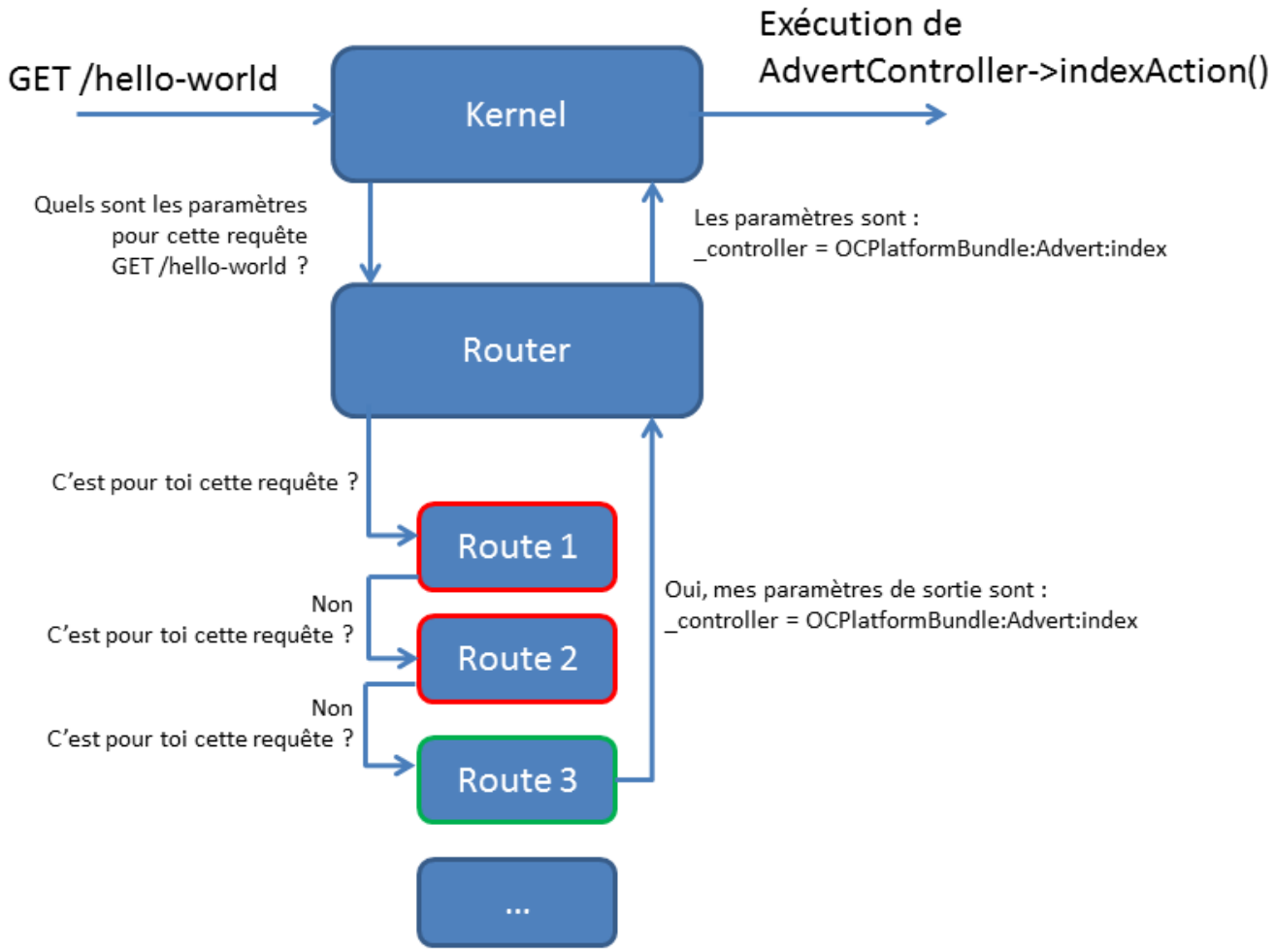
- Rappel : toute requête arrive sur le « Front Controller » (unique point d'entrée de l'application)

- Chaque requête doit être acheminée vers le code PHP qui sait la traiter : c'est le routing.



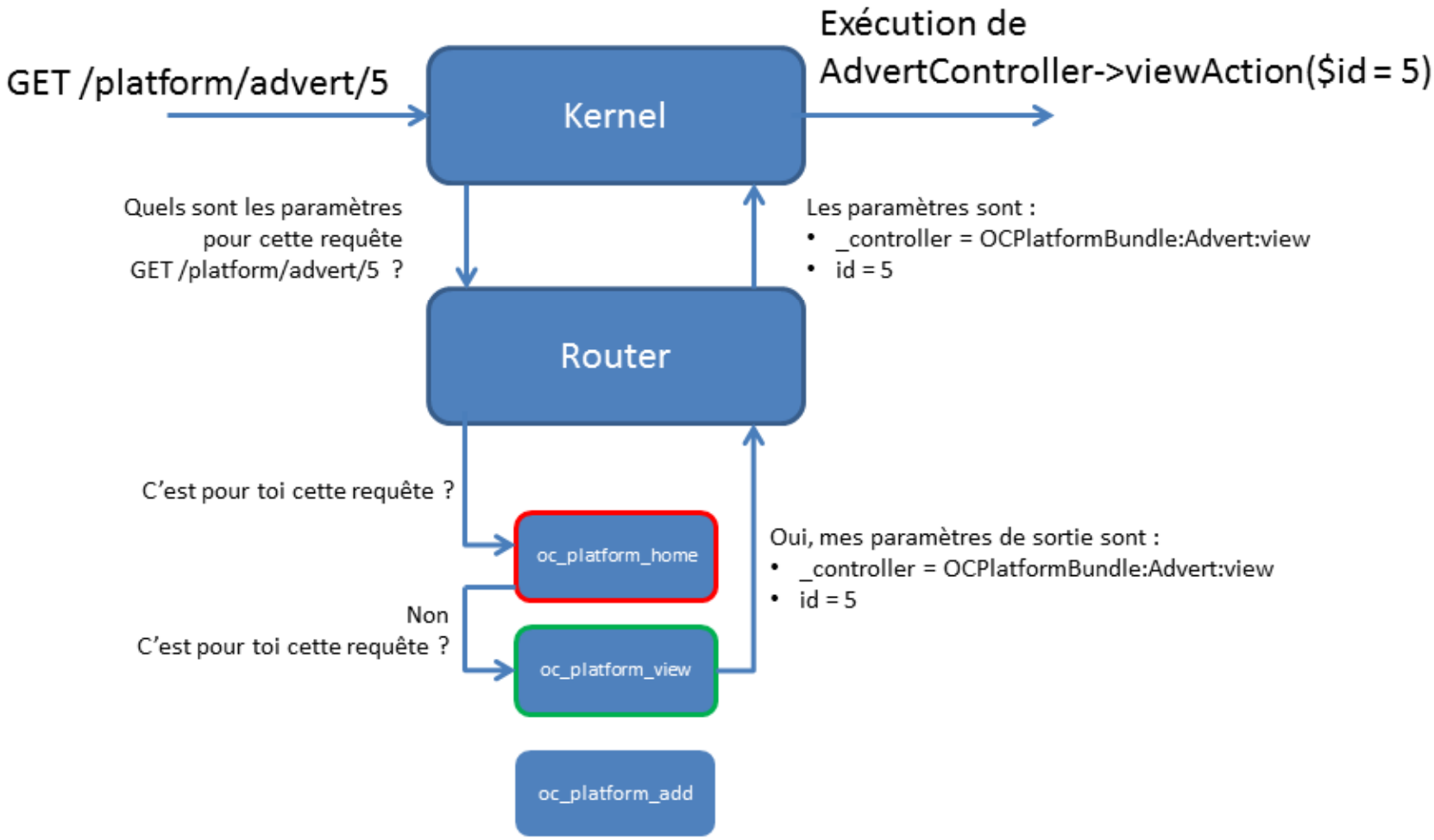
Symfony en pratique

Le routing, concrètement



Symfony en pratique

Le routing, concrètement



Symfony en pratique

Le routing, concrètement

- **app/config/routing.yml** (fichier global à l'application)

```
oc_blog:
    resource: "@OCBlogBundle/Resources/config/routing.yml"
    prefix:   /blog
```

- **src/OC/BlogBundle/Resources/config/routing.yml** (fichier local au bundle)

```
oc_blog_hello:
    path:     /hello
    defaults: { _controller: OCBlogBundle :Default:index"}

oc_blog_home:
    path:     /
    defaults: { _controller: OCBlogBundle:Film:index" }
```

Symfony en pratique

Le routing : comprendre les routes

- Pour savoir à quel contrôleur Symfony va envoyer la requête, il va consulter le fichier de configuration du routing du bundle
  - src\OCBlogBundle\Resources\config\routing.yml

Nom interne de la route

Masque d'URL correspondant

Code PHP à appeler pour fabriquer la réponse

```
oc_blog_hello :
    path:     /hello
    defaults: { _controller: OCBlogBundle:Default:index }
```

« **OCBlogBundle:Defaut:index** » est le nom logique du contrôleur

Méthode **indexAction** de la classe **DefaultController**

Symfony en pratique

Le routing, par annotations

```yaml
blog:
    path:       /blog
    defaults:   { _controller: AppBundle:Blog:index }
```

=

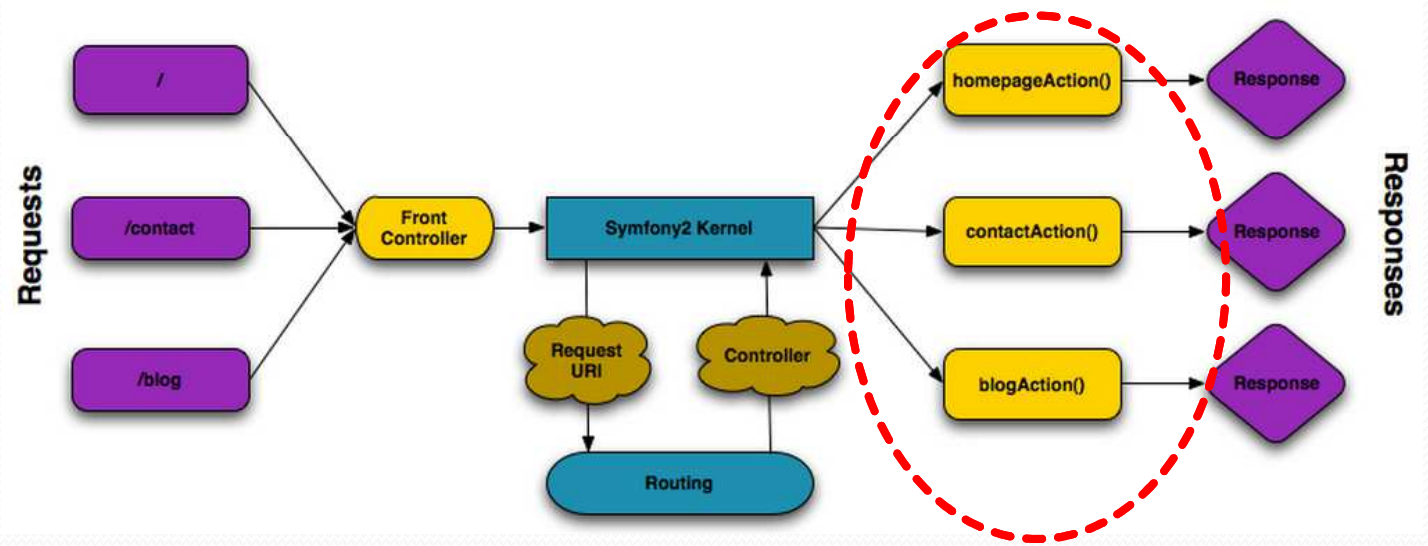
```php
class BlogController extends Controller
{
    // ...

    /**
     * @Route("/blog")
     */
    public function indexAction()
    {
        // ...
    }
}
```

Symfony en pratique

Ecrire un contrôleur

- Maintenant que le framework sait vers quelle méthode PHP diriger la requête, il faut écrire cette méthode



- Le but d'un contrôleur est de créer une **réponse** à renvoyer à l'utilisateur, à partir des paramètres fournis dans la **requête**

Symfony en pratique

Ecrire un contrôleur pour une route /test/hello/{name}

- Pour l'exemple :

```
test_hello:
    path:     /test/hello/{name}
    defaults: { _controller: OCBlogBundle:Default:hello }
```

  - La classe du contrôleur associée à cette route se trouve dans :
    - OC/BlogBundle/Controller/DefaultController.php
  - La méthode à écrire est :
    - helloAction
- Comme la route comprend un paramètre, la méthode aura également un paramètre :
    - public function helloAction($name)

Symfony en pratique

Ecrire un contrôleur pour la route /test/hello/{name}

- Que faire dans la méthode helloAction ?
  - Renvoyer directement du HTML ?
    - Symfony le permet, puisqu'il est possible de fabriquer de toute pièce une réponse

```
public function helloAction($name)
{
    return new Response('<html><body>Bonjour '.$name.'!</body></html>');
}
```

A ne pas faire !!!

Symfony en pratique

Ecrire un contrôleur pour la route /test/hello/{name}

- Que faire dans la méthode helloAction ?
  - Renvoyer le **rendu d'un template**
    - C'est ce que Symfony fait dans la méthode d'exemple générée lors de la création du bundle

```
public function helloAction($name) {
    return $this->render('OCBlogBundle:Default:hello.html.twig'
                        , array('name' => $name));
}
```

Pour utiliser la méthode **render()**,

- le contrôleur doit hériter de **Controller** (extends …)
- la classe Controller doit avoir été importée (use …)

Symfony en pratique

Le moteur de rendu de templates

- La méthode **render()** fait en réalité appel au moteur de rendu de templates (**Twig** en l'occurrence) utilisé sur le projet

- Le chemin du template est une fois encore un chemin « logique » :

```
OCBlogBundle:Default:hello.html.twig
```

OC/BlogBundle/ resources/views/**Default** **hello.html.twig**

- Caractéristiques d'un moteur de templates

  - **Prend en entrée un template**, c'est-à-dire un modèle de page
    - qui contient des variables, des instructions, des boucles, des conditions…
    - qui respecte une certaine syntaxe (= langage à part entière)

  - **Génère un contenu HTML** à partir de ce modèle et des données nécessaires fournies

Symfony en pratique

Le moteur de templates « Twig »

- Simple mais puissant
- 3 types de syntaxes :
  - {{ … }} : écrit la valeur d'une variable ou d'une expression dans le template
  - {% … %} : exécute une instruction (boucle, …)
    - Liste des tags disponibles : http://twig.sensiolabs.org/doc/tags/index.html
  - {# … #} : ajoute un commentaire
- Possibilité d'utiliser des filtres
  - {{ mavariable | upper }} : écrit la variable mavariable en majuscules
    - Liste des filtres disponibles : http://twig.sensiolabs.org/doc/filters/index.html

Symfony en pratique

Le moteur de templates « Twig »

- Exemple :
  - Le template correspondant à notre méthode helloAction :

```
<!DOCTYPE html>
<html>
    <head>
        <title>My test hello page !</title>
    </head>
    <body>
        <h1>Bonjour {{ name }}</h1>
    </body>
</html>
```

<u>Problème :</u>
Souvent, des éléments sont communs à plusieurs templates, il faudra les dupliquer ➔ Non satisfaisant !

Symfony en pratique

Le moteur de templates « Twig »

- Twig offre plusieurs méthodes pour éviter la duplication de code dans les templates
  - L'héritage de template,
  - La surcharge de template
  - L'inclusion de template,
  - L'imbrication d'un contrôleur dans un template
- Ces différentes techniques sont basées sur le concept de

Symfony en pratique

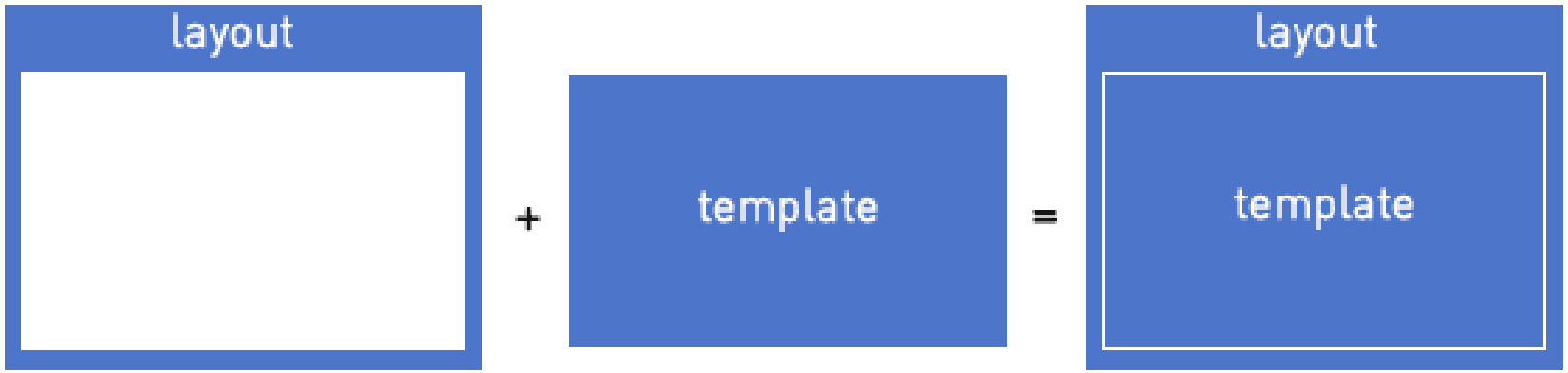
Les Layouts

- Le concept de **layout**
  - Définition : une disposition de page (=mise en page)
    - Dispose les éléments sur la page
    - Évite la duplication de code pour les éléments communs à toutes les pages
  - Première solution

Symfony en pratique

Les Layouts

- Le concept de **layout**
  - Deuxième, et bonne solution : le pattern Decorator



Symfony en pratique

Les Layouts en pratique

- Reprenons notre exemple
  - Pour obtenir notre page, on crée

```
<!DOCTYPE html>
<html>
    <head>
        <title>My test hello page !</title>
    </head>
    <body>
        <h1>Bonjour {{ name }}</h1>
    </body>
</html>
```

1 Un template (Layout) de base, dans lequel on va définir des blocs

- app/resources/views/base.html.twig

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        <title>{% block title %}Welcome!{% endblock %}</title>
    </head>
    <body>
        {% block body %}{% endblock %}
    </body>
</html>
```

Déclaration d'un bloc :

{% block <nom>} … {% endblock %}

Symfony en pratique

Les Layouts en pratique

- Reprenons notre exemple
  - Pour obtenir notre page, on crée

```
<!DOCTYPE html>
<html>
    <head>
        <title>My test hello page !</title>
    </head>
    <body>
        <h1>Bonjour {{ name }}</h1>
    </body>
</html>
```

2 Un template enfant, qui étend le layout de base, et qui surcharge les blocs souhaités

- OC/BlogBundle/Resources/views/Default/hello.html.twig

```
{% extends "::base.html.twig" %}
{% block title %}My test hello page !{% endblock %}
{% block body %}
    <h1>Bonjour pour de vrai {{ name }}</h1>
{% endblock %}
```

Surcharge d'un bloc :

{% block <nom>} … {% endblock }

Symfony en pratique

Les Layouts en pratique

- Cette solution permet facilement de :
  - Conserver le contenu d'un bloc défini dans le layout de base
    - Exemple : je ne définis pas le bloc « title » dans mon template enfant.
  - Remplacer le contenu d'un bloc défini dans le layout de base
    - Exemple : ce qu'on a fait précédemment.
- Il est également possible de « compléter » un bloc défini dans le layout de base
    - Exemple : utilisation de la fonction {{ parent() }}

```
{% extends "::base.html.twig" %}
{% block title %}{{ parent() }} Bienvenue! Bienvenido!{% endblock %}
{% block body %}
    <h1>Bonjour pour de vrai {{ name }}</h1>
{% endblock %}
```

Symfony en pratique

Inclusion de CSS, Javascript et images avec Twig

- L'inclusion de ressources (CSS, JS..) profite largement du mécanisme d'héritage de templates
- Définition dans le layout de base
  - d'un bloc pour l'inclusion des CSS (dans le HEAD)
  - d'un bloc pour l'inclusion des JS (à la fin du BODY)

Symfony en pratique

Inclusion de CSS, Javascript et images avec Twig

- Exemple dans le layout de base

```
{# src/OC/BlogBundle/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 Blog{% endblock %}</title>
        {% block stylesheeticon %}
            <link rel="stylesheet" href="{{ asset('css/bootstrap.css') }}" />
        {% endblock %}
    </head>
    <body>
    {% block body %}{% endblock %}
    {% block javascripts %}
        <script src="{{ asset('js/jquery.min.js') }}"></script>
    {% endblock %}
</body>
</html>
```

Symfony en pratique

Inclusion de CSS, Javascript et images avec Twig

- Exemple dans un template enfant

```
{# src/OC/BlogBundle/Resources/views/Default/asset.html.twig #}
{% extends "OCBlogBundle::layout.html.twig" %}
{% block title %}{{ parent() }} - Animation{% endblock %}
{% block stylesheeticon %} {{ parent() }}
    <link rel="icon" href="{{ asset('images/favicon.ico') }}" />
{% endblock %}
{% block body %} {# ... #} {% endblock %}
{% block javascripts %} {{ parent() }}
    <script src="{{ asset('js/couleur.js') }}"></script>
{% endblock %}
```

- Le mécanisme d'héritage permet d'ajouter facilement des ressources spécifiques au template enfant

- On remarque l'utilisation de la fonction « asset() », qui permet de calculer automatiquement le chemin réel de la ressource

Symfony en pratique

Inclusion de CSS, Javascript et images avec Twig

Emplacement des ressources

- Les ressources sont placées au niveau « global » du site (solution simple, à privilégier pour la plupart des développements). Les dossiers suivants sont à créer :
  - web\css
  - web\images
  - web\js

- Une autre solution, plus « propre », mais beaucoup plus contraignante, est de placer les ressources relatives à un bundle à l'intérieur de ce bundle (non conseillé pour un projet simple).

Symfony en pratique

Création de liens entre pages avec Twig

- On ne code pas les URL « en dur » dans les templates
- Les URLs doivent être générées par le framework
  - Utilise le mécanisme de routing
  - Permet en cas de changement d'une URL de ne pas avoir à la modifier dans plusieurs templates
- Génération d'URL relatives et absolues

Symfony en pratique

Création de liens entre pages avec Twig

- Utilisation de la fonction path de Twig
  - Route sans paramètre

```
oc_blog_home:
    path:     /
    defaults: { _controller: OCBlogBundle:Film:index }
```

  - Création d'un lien vers cette route

```
<a href="{{ path('oc_blog_home') }}" class="btn btn-default">
    <i class="glyphicon glyphicon-chevron-left"></i>
    Retour à la liste des films
</a>
```

Symfony en pratique

Création de liens entre pages avec Twig

- Utilisation de la fonction path de Twig
  - Route avec paramètre

```
oc_blog_view:
    path:          /film/{id}
    defaults:      { _controller: OCBlogBundle:Film:view }
    requirements:  { id: \d+ }
```

-

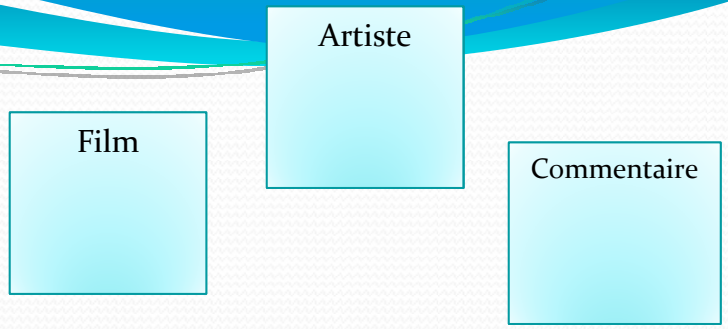
```
<a href="{{ path('oc_blog_view', {'id': film.id}) }}">
    <em>Commentaires</em>
</a>
```

Ici, La variable film est une variable connue du template (soit le film courant d'une boucle, soit un paramètre qui a été passé au template…)

- De la même manière, la fonction url permet de générer des URL absolues

Symfony en pratique

Les données



- Données métier manipulées = « Modèle » du MVC
- Stockage des données en base de données
- Réalisation d'une connexion « à la main » en PHP

```
$connexion = new PDO('mysql:host='.SERVEUR.';dbname='. BDD.';charset=utf8'
        , USER, PASSWORD
        , array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
```

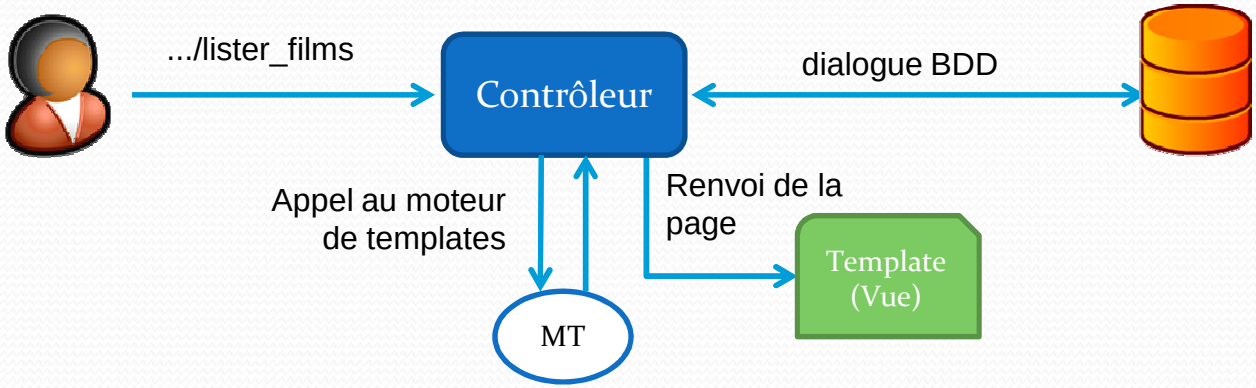
Ouvertures / Fermetures de connexions

- Création de requêtes

```
    $requete = <<<SQL
        SELECT titre, annee, genre, nom, prenom, anneeNaiss
        FROM Film JOIN Artiste ON idMES = idArtiste
SQL;
    $resultat = $connexion->prepare($requete);
    $resultat->execute();
    while ($film = $resultat->fetch(PDO::FETCH_ASSOC)):
        // ...
    endwhile;
```

Symfony en pratique

Accès aux données : mauvaise solution



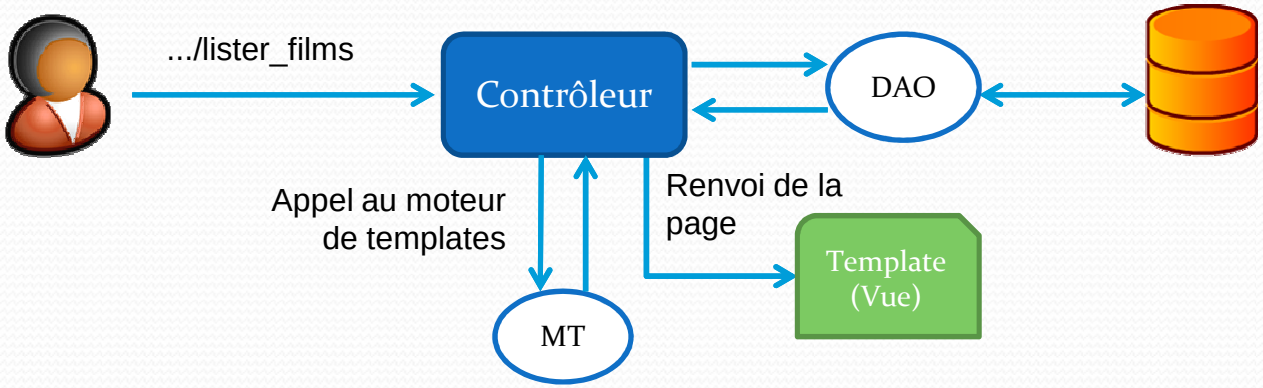
- Le contrôleur peut réaliser :
  - L'ouverture de connexion à la BDD
  - Les requêtes nécessaires pour récupérer les données
  - La fermeture de la connexion

- Inconvénients :
  - Aucun code réutilisable
  - Beaucoup de duplication de code
  - Maintenance difficile

Symfony en pratique

Accès aux données : bonne solution

- Idée :



- Le contrôleur peut faire appel à un composant dédié (DAO pour Data Access Object), chargé du dialogue avec la BDD
  - Code réutilisable : le DAO peut être appelé par plusieurs contrôleurs différents
  - Duplication : le code est factorisé dans le DAO
  - Maintenance : elle est facilitée (chaque composant à son rôle, et on ne duplique plus)

On a toujours un problème :
ouvertures et fermetures de connexions…

Symfony en pratique

Accès aux données : les outils Symfony

- Symfony fournit plusieurs outils :
  - Un outil très simple : Doctrine DBAL (seul)
  - Un outil complet mais difficile à maitriser : Doctrine ORM (complet)

Symfony en pratique

Accès aux données : Doctrine DBAL

- Doctrine DBAL (DataBase Abstraction Layer)
  - Couche d'abstraction de la base de données
  - API flexible pour communiquer avec la base de données
    - Permet la connexion à toutes les BD relationnelles les plus connues
    - Facilite la configuration de la connexion,
    - Facilite la connexion et l'exécution de requêtes

Symfony en pratique

Paramétrage et utilisation de Doctrine DBAL

- 2 choses à faire pour utiliser Doctrine DBAL
  - Fournir les paramètres de connexion à la base dans la config de Symfony
  - Récupérer une instance de connexion à la base lorsque l'on en a besoin

1 - Fourniture des paramètres

- Fichier : app/config/parameters.yml
- et app/config/config.yml

```
parameters:
    database_host: 127.0.0.1
    database_port: null
    database_name: oc_blog
    database_user: root
    database_password: null
```

```
imports:
    - { resource: parameters.yml }
#...
# Doctrine Configuration
doctrine:
    dbal:
        driver:   pdo_mysql
        host:     "%database_host%"
        port:     "%database_port%"
        dbname:   "%database_name%"
        user:     "%database_user%"
        password: "%database_password%"
        charset:  UTF8
```

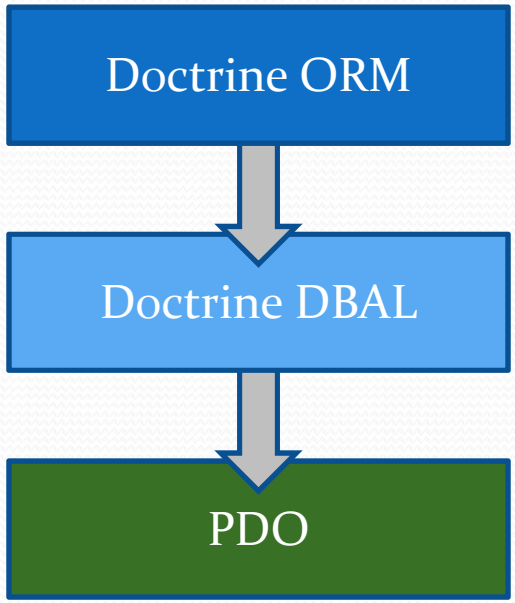
2 - Récupération d'une instance de connexion à la base

- Dans un contrôleur qui étend la classe **Controller** :

```
$this->get('database_connection');
```

Symfony en pratique

Accès aux données : L'ORM Doctrine



- S'appuie sur la couche DBAL
- Permet d'utiliser un langage de requêtes orienté objet
- Permet de ne manipuler que des objets

Symfony en pratique

Utilisation de Doctrine ORM

Rappel

- Paramétrage de la connexion à la base dans : app/config/parameters.yml

Première méthode (manuelle) de création des entités

- Création de la base de données (BDD) :

php bin/console doctrine:database:create

(Cette commande réalise simplement le CREATE DATABASE)

- Création des entités du modèle de données :

php bin/console doctrine:generate:entity

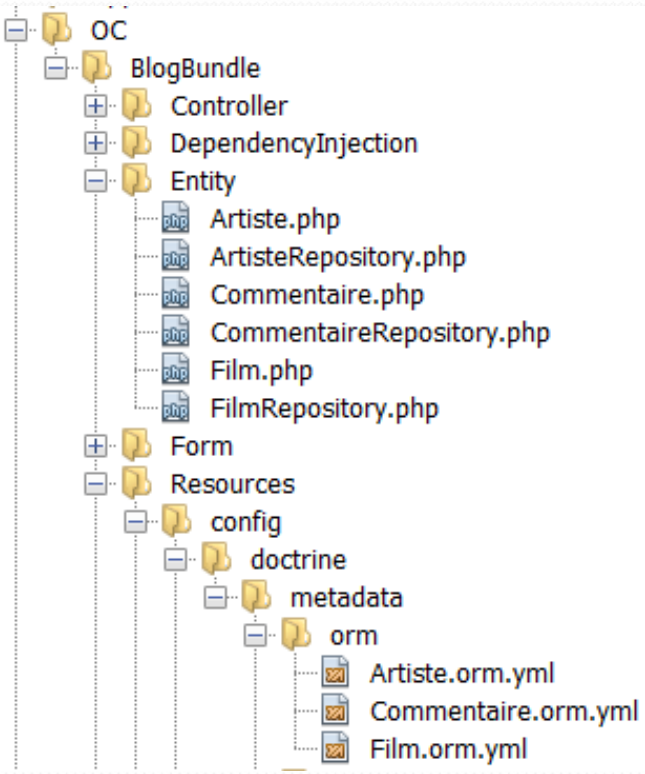
(La commande automatise l'écriture d'une classe du modèle : demande le nom de la classe, le type des champs, et le format de la configuration de mapping. Elle propose la génération d'une classe « Repository » vide associée, ce qu'il est conseillé de faire)

Symfony en pratique

Utilisation de Doctrine ORM

Une **entité** est une classe du modèle de données, qui servira à persister des instances d'objets dans une table de la base.

class Film



Pour chaque entité gérée par l'ORM, il faut :

- Créer la classe PHP de l'entité (Ex : Film.php)
- Créer le fichier de mapping si on a choisi le format YML (Ex : Film.orm.yml)
- Créer la classe PHP du Repository (Ex : FilmRepository.php)

Symfony en pratique

Utilisation de Doctrine ORM

Préalables

- Création de la BDD via phpMyAdmin
- Paramétrage de la connexion à la base dans : app/config/parameters.yml

Deuxième méthode de création des entités par rétro-conception

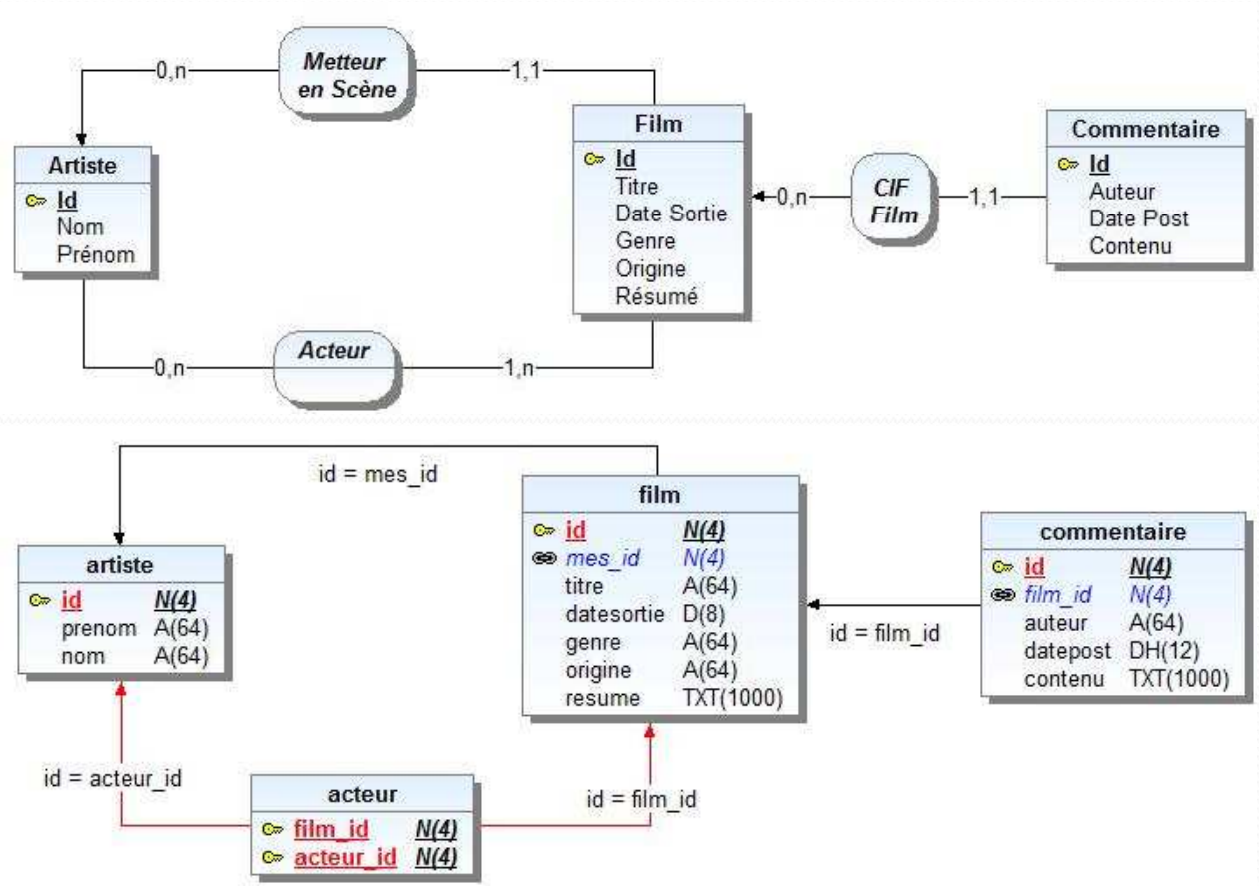
- Création classique des tables à partir du MCD (cf. Win'Design et phpMyAdmin)

MCD ⟶ MLDR ⟶ MPDR ⟶ Création des tables via phpMyAdmin

Symfony en pratique

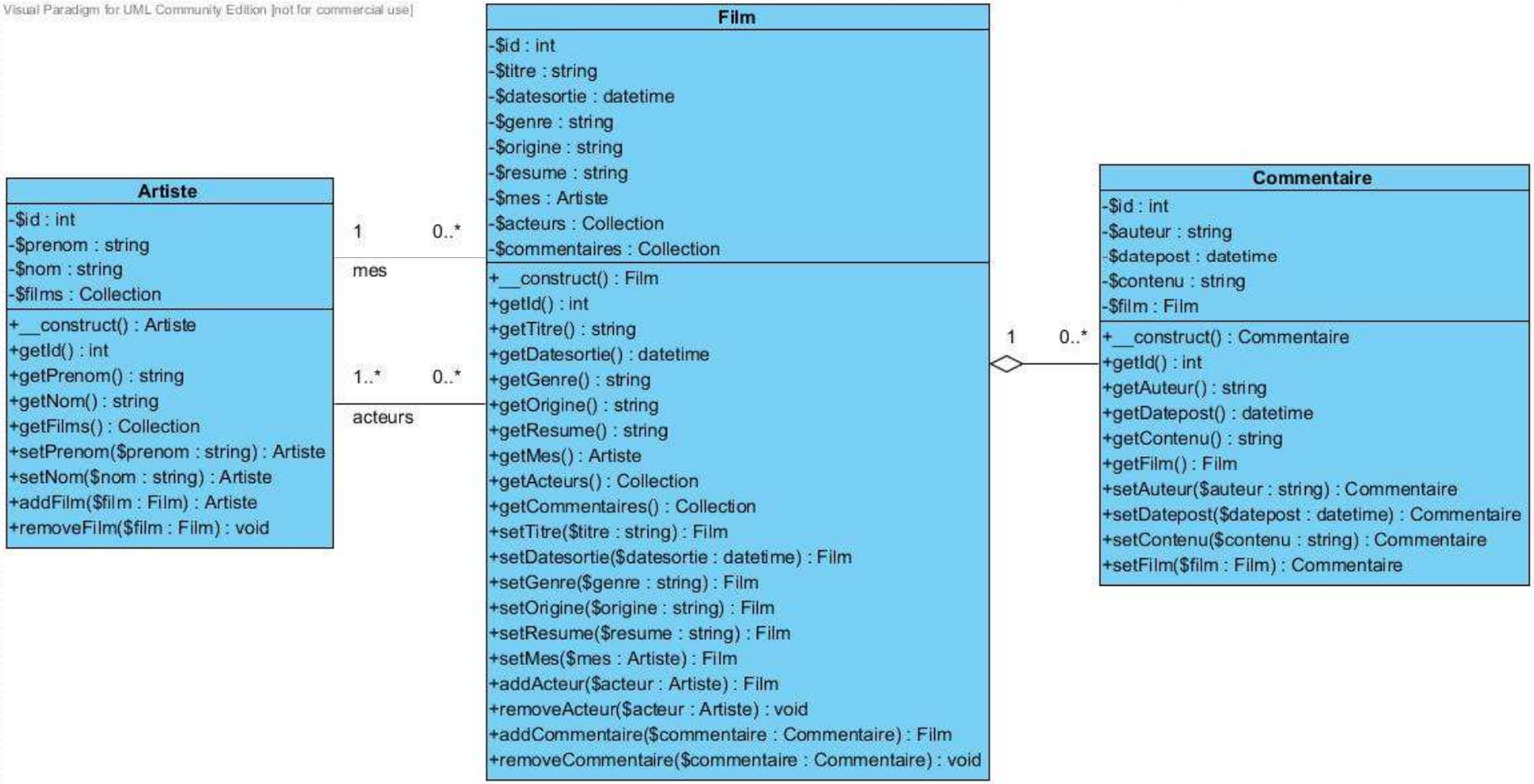
Utilisation de Doctrine ORM

- MCD et MLDR et Diagramme de classes du cas Blog_Films :



Symfony en pratique

Utilisation de Doctrine ORM



Symfony en pratique

Utilisation de Doctrine ORM

- Extrait du MPDR du cas Blog_Films :

```
CREATE TABLE film (
    id              INT(4) NOT NULL AUTO_INCREMENT
    , mes_id        INT(4) NOT NULL
    , titre         VARCHAR(64)
    , datesortie    DATE
    , genre         VARCHAR(64)
    , origine       VARCHAR(64)
    , resume        LONGTEXT
    , CONSTRAINT PK_Film PRIMARY KEY (id)
);
CREATE TABLE commentaire (
    id              INT(4) NOT NULL AUTO_INCREMENT
    , film_id       INT(4) NOT NULL
    , auteur        VARCHAR(64)
    , datepost      DATETIME
    , contenu       LONGTEXT
    , CONSTRAINT PK_Commentaire PRIMARY KEY (id)
);
ALTER TABLE commentaire  ADD (
    CONSTRAINT FK_Commentaire_Film
        FOREIGN KEY (film_id)
        REFERENCES film (id) );
```

Symfony en pratique

Utilisation de Doctrine ORM

- Demander à **Doctrine** d'introspecter la BDD et de générer les fichiers de méta-données correspondants en yml :

php bin/console doctrine:mapping:convert yml ./src/OC/BlogfilmBundle/Resources/config/doctrine/metadata/orm --from-database --force

```
Film:
    type: entity
    table: film
    indexes:
        FK_Film_Artiste:
            columns:
                - mes_id
    id:
        id:
            type: integer
            nullable: false
            options:
                unsigned: false
            id: true
            generator:
                strategy: IDENTITY
```

```
    fields:
        titre:
            type: string
            nullable: true
            length: 64
            options:
                fixed: false
        datesortie:
            type: date
            nullable: true
        genre:
            type: string
            nullable: true
            length: 64
            options:
                fixed: false
```

Symfony en pratique

Utilisation de Doctrine ORM

- Demander à **Doctrine** d'importer les schémas yml et de construire les classes entité (**sans les getters ni les setters**) qui leur sont liées :

  php bin/console doctrine:mapping:import OCBlogfilmBundle annotation

- Rajouter des annotations ORM ou des attributs non générées automatiquement telles que :
  - *La définition d'un repository*
  - *La définition d'attributs non persités en BDD*
  - *La définition d'un constructeur incluant des initialisations d'attributs dont ceux qui font l'objet d'annotation ORM de type OneToMany ou ManyToMany qui représentent des objets de type ArrayCollection*

- Modifier certaines annotations générées automatiquement telles que :
  - *IDENTITY en AUTO*
  - *Faire le tour des relations (OneToMany, ManyToMany, ManyToOne) entre entités en rajoutant, si besoin est, des options telles que inversedBy, mappedBy, cascade, etc.*

Symfony en pratique

Utilisation de Doctrine ORM

```php
<?php
namespace OC\BlogfilmBundle\Entity;
use Doctrine\Common\Collections\ArrayCollection;
use Doctrine\ORM\Mapping as ORM;

/**
 * Film
 *
 * @ORM\Table(name="film", indexes={@ORM\Index(name="FK_Film_Artiste", columns={"mes_id"})})
 * @ORM\Entity(repositoryClass="OC\BlogfilmBundle\Entity\FilmRepository")
 */
class Film {

    /**
     * @var integer
     *
     * @ORM\Column(name="id", type="integer", nullable=false)
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;
```

Symfony en pratique

Utilisation de Doctrine ORM

```
/**
 * @var string
 *
 * @ORM\Column(name="titre", type="string", length=64, nullable=true)
 */
private $titre;

/**
 * @var \DateTime
 *
 * @ORM\Column(name="datesortie", type="date", nullable=true)
 */
private $datesortie;

/**
 * @var string
 *
 * @ORM\Column(name="genre", type="string", length=64, nullable=true)
 */
private $genre;
```

Symfony en pratique

Utilisation de Doctrine ORM

```
/**
 * @var \Artiste
 *
 * @ORM\ManyToOne(targetEntity="OC\BlogfilmBundle\Entity\Artiste")
 * @ORM\JoinColumns({
 *   @ORM\JoinColumn(name="mes_id", referencedColumnName="id")
 * })
 */
private $mes;

/**
 * @var \Doctrine\Common\Collections\Collection
 *
 * @ORM\ManyToMany(targetEntity="OC\BlogfilmBundle\Entity\Artiste", inversedBy="films“
 * , cascade={"persist"})
 * @ORM\JoinTable(name="acteur",
 *   joinColumns={
 *     @ORM\JoinColumn(name="film_id", referencedColumnName="id")
 *   },
 *   inverseJoinColumns={
 *     @ORM\JoinColumn(name="acteur_id", referencedColumnName="id")
 *   }
 * )
 */
private $acteurs;
```

Symfony en pratique

Utilisation de Doctrine ORM

```
/**
 * @ORM\OneToMany(targetEntity="OC\BlogfilmBundle\Entity\Commentaire“
 * , mappedBy="film", cascade={"remove"})
 */
private $commentaires;

/**
 * Constructor
 */
public function __construct() {
    $this->acteurs = new ArrayCollection();
    $this->commentaires = new ArrayCollection();
}
```

Symfony en pratique

Utilisation de Doctrine ORM

- Générer les getters et les setters après toute modification d'entités :

  php bin/console doctrine:generate:entities OCBlogfilmBundle:EntityName

  etc.

- Répercuter toute modification d'entité en BDD :

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

- Et si au moins une requête SQL est générée par Doctrine, lancer la commande suivante pour exécuter les requêtes générées par Doctrine :

  php bin/console doctrine:schema:update --force

Symfony en pratique

Utilisation de Doctrine ORM

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

    /**
     * Set titre
     *
     * @param string $titre
     *
     * @return Film
     */
    public function setTitre($titre) {
        $this->titre = $titre;

        return $this;
    }
```

Symfony en pratique

Utilisation de Doctrine ORM

```
/**
 * Set datesortie
 *
 * @param \DateTime $datesortie
 *
 * @return Film
 */
public function setDatesortie($datesortie) {
    $this->datesortie = $datesortie;

    return $this;
}

/**
 * Get datesortie
 *
 * @return \DateTime
 */
public function getDatesortie() {
    return $this->datesortie;
}
```

Symfony en pratique

Utilisation de Doctrine ORM

```
    /**
     * Set mes
     *
     * @param \OC\BlogBundle\Entity\Artiste $mes
     *
     * @return Film
     */
    public function setMes(\OC\BlogfilmBundle\Entity\Artiste $mes) {
        $this->mes = $mes;

        return $this;
    }

    /**
     * Get mes
     *
     * @return \OC\BlogfilmBundle\Entity\Artiste
     */
    public function getMes() {
        return $this->mes;
    }
```

Symfony en pratique

Utilisation de Doctrine ORM

```
    /**
     * Add acteur
     *
     * @param \OC\BlogBundle\Entity\Artiste $acteur
     *
     * @return Film
     */
    public function addActeur(\OC\BlogfilmBundle\Entity\Artiste $acteur) {
        $this->acteurs[] = $acteur;
        return $this;
    }
    /**
     * Remove acteur
     *
     * @param \OC\BlogBundle\Entity\Artiste $acteur
     */
    public function removeActeur(\OC\BlogfilmBundle\Entity\Artiste $acteur) {
        $this->acteurs->removeElement($acteur);
    }
    /**
     * Get acteurs
     *
     * @return \Doctrine\Common\Collections\Collection
     */
    public function getActeurs() {
        return $this->acteurs;
    }
```

Symfony en pratique

Utilisation de Doctrine ORM

Un **repository** (ou dépôt) est une classe PHP dont le seul rôle est de permettre la récupération des entités d'un type particulier (Film, Commentaire,…)

class FilmRepository **extends** EntityRepository

Plusieurs méthodes sont prêtes à être utilisées (car héritées de EntityRepository) :

- find($id)
- findOneById($id)
- findOneByName($name)
- findAll()
- findByDate($date)

- FindOneBy<Champ>
- findBy<Champ>

Sont des méthodes présentes automatiquement en fonctions des champs de l'entité

Symfony en pratique

Utilisation de Doctrine ORM

- Doctrine peut générer pour chaque entité ce que l'on appelle un CRUD
  - Page d'affichage des objets du modèle
  - Page de création d'un nouvel objet
  - Page d'édition d'un objet
  - Page de suppression d'un objet

- Génération d'un CRUD pour une entité

php bin/console generate:doctrine:crud

Symfony en pratique

Utilisation de Doctrine ORM

- Création / mise à jour du schéma dans la base de données

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

php bin/console doctrine:schema:update –force

(Cette commande détecte automatiquement les changements à effectuer dans la base, en analysant les entités du modèle de données !)

- Après ajout d'une nouvelle entité ou modification d'une entité existante, il faut penser à re-exécuter la commande de mise à jour du schéma de BDD

Symfony en pratique

Utilisation de Doctrine ORM

- Enregistrement d'un objet en BDD
  - Par exemple, dans la méthode d'action gérant l'ajout d'un commentaire (dans un contrôleur)

```
$em = $this->getDoctrine()->getManager();
$em->persist($commentaire);
$em->flush();
```

- Récupération d'objets depuis la BDD

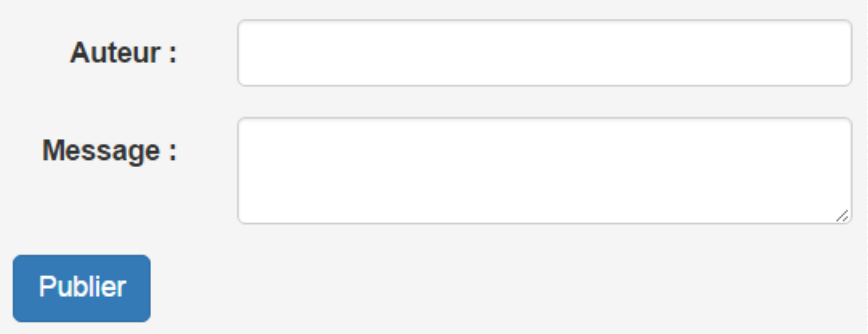
```
$em = $this->getDoctrine()->getManager();
$listFilms = $em
        ->getRepository('OCBlogBundle:Film')
        ->getFilms()
;
```

- Suppression d'objet dans la BDD

```
$em->remove($advert);
$em->flush();
```

Symfony en pratique

Les formulaires



- Permettent de récupérer les données saisies par l'utilisateur
- Souvent une corvée pour le développeur
  - Beaucoup de code
  - Des règles de validation pour chaque champ
  - Des messages d'erreurs à afficher en cas de champ incorrect
  - Ne pas perdre les valeurs saisies en cas de demande de modification d'un champ

Symfony en pratique

Création d'un formulaire

- La vision Symfony sur les formulaires est la suivante : un formulaire se construit sur un objet existant, et son objectif est d'hydrater cet objet.

- Avec Symfony, pour créer un formulaire, il nous faut :
  - Un objet
  - Un moyen pour construire un formulaire à partir de cet objet, un `FormBuilder` (« constructeur de formulaire »)

- Les principales étapes de création de formulaire sont :
  - Création d'une instance de l'objet concerné par le formulaire
  - Création du `FormBuilder` en le liant à l'instance de l'objet grâce au service « `form factory` » et rajout de champs de l'objet grâce à la méthode `add()`
  - Création du formulaire à partir du FormBuilder
  - Affichage du formulaire par la vue grâce à la méthode `createView()` du formulaire

Symfony en pratique

Constructeur de formulaire : première approche non satisfaisante

```
// use des nemaspaces des Entités utilisées
// use des Types de champs du formulaire
public function addAction(Request $request) {
    $advert = new Advert();// On crée un objet Advert
    // On crée le FormBuilder grâce au service form factory
    $formBuilder = $this->get('form.factory')
                        ->createBuilder(FormType::class, $advert);
    $formBuilder
            ->add('date', DateType::class)
            ->add('title', TextType::class)
            ->add('content', TextareaType::class)
            ->add('author', TextType::class)
            ->add('published', CheckboxType::class)
            ->add('save', SubmitType::class);
        // On génère le formulaire à partir du formBuilder
    $form = $formBuilder->getForm();
    return $this->render('OCPlatformBundle:Advert:add.html.twig'
                        , array('form' => $form->createView()));
}
```

Symfony en pratique

Rendu d'un formulaire

- Le formulaire devient un objet « affichable » par le template `add.html.twig` grâce à l'application de la méthode `createView()` sur l'objet formulaire généré par le FormBuilder.

```
    return $this->render('OCPlatformBundle:Advert:add.html.twig', array(
            'form' => $form->createView()
));
```

- Le rendu complet du formulaire devient effectif grâce à l'inclusion dans `add.html.twig` du template `form.html.twig` réutilisable suivant.

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

<div class="well">
    {{ form(form) }}
</div>
```

Symfony en pratique

```
{# src/OC/PlatformBundle/Resources/views/Advert/add.html.twig #}
{% block title %}
    Ajouter une annonce - {{ parent() }}
{% endblock %}
{% block ocplatform_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>
    <p>
        <a href="{{ path('oc_platform_home') }}" class="btn btn-default">
            <i class="glyphicon glyphicon-chevron-left"></i>
            Retour à la page d'accueil
        </a>
    </p>
{% endblock %}
```

```
{# src/OC/PlatformBundle/Resources/views/Advert/form.html.twig #}
<h3>Formulaire d'annonce</h3>
<div class="well">
    {{ form(form) }}
</div>
```

Symfony en pratique

Les formulaires : mauvais exemple !

- L'exemple précédent ne respecte pas le principe de séparation de code ! Chacun son rôle !
- Le contrôleur ne devrait pas contenir toute la logique de création du formulaire
- Si un autre contrôleur nécessite le même formulaire, on doit faire du copier/coller
- Si l'on doit ajouter un champ à un formulaire, on doit l'ajouter partout où l'on a inséré la construction du formulaire.

Symfony en pratique

Externaliser la création du constructeur de formulaire

- Commande de génération de constructeur de formulaire :
  php bin/console doctrine:generate:form OCBlogBundle:EntityName
  etc.

- Le constructeur de formulaire nommé *EntityNameType* est généré dans le répertoire :
  - **src/OC/BlogBundle/Form/**

- Modifier *EntityNameType* en typant les champs qui représentent des attributs retenus de *EntityName*

Symfony en pratique

```php
<?php
namespace OC\PlatformBundle\Form;

use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\OptionsResolver\OptionsResolver;
use Symfony\Component\Form\Extension\Core\Type\CheckboxType;
// use des autres Types de champs du formulaire
class AdvertType extends AbstractType {

    public function buildForm(FormBuilderInterface $builder, array $options) {
        $builder
                ->add('date', DateTimeType::class
                     , array('widget' => 'single_text'
                           , 'format' => 'dd-MM-yyyy HH:mm:ss'))
                ->add('title', TextType::class)
                ->add('author', TextType::class)
                ->add('content', TextareaType::class)
                ->add('published', CheckboxType::class)
                ->add('save', SubmitType::class);
    }

    public function configureOptions(OptionsResolver $resolver) {
        $resolver->setDefaults(array(
            'data_class' => 'OC\PlatformBundle\Entity\Advert'
        ));
    }

}
```

Symfony en pratique

Gestion de la soumission d'un formulaire

Lors de la soumission d'un formulaire par un utilisateur, il faut gérer l'envoi du formulaire :

- Il faut tout d'abord vérifier que la requête est de type **POST** signifiant que l'utilisateur vient de soumettre le formulaire.
- Il faut ensuite que l'objet formulaire fasse appel à la méthode handleRequest() afin d'hydrater l'objet sous-jacent.
- Il faut enfin que l'objet formulaire fasse appel à la méthode isValid() sur l'objet hydraté afin de vérifier la validation des données saisies via le formulaire avant que l'hydratation de l'objet ne devienne définitive.

Symfony en pratique

Constructeur de formulaire : la bonne approche

```
namespace OC\PlatformBundle\Controller;

use OC\PlatformBundle\Entity\Advert;
use OC\PlatformBundle\Form\AdvertType;

public function addAction(Request $request) {
        $advert = new Advert(); // On crée un objet Advert
    // On crée le Formulaire grâce au service form factory
    $form = $this->get('form.factory')->create(AdvertType::class, $advert);

    // Si la requête est en POST et les valeurs entrées sont correctes
    if ($request->isMethod('POST') && $form->handleRequest($request)->isValid()) {
        $em = $this->getDoctrine()->getManager();
        $em->persist($advert);
        $em->flush();
        return $this->redirectToRoute('oc_platform_view‘
                                    , array('id' => $advert->getId()));
    }
    // on affiche le formulaire
    return $this->render('OCPlatformBundle:Advert:add.html.twig'
                        , array('form' => $form->createView()));
}
```

Symfony en pratique

Template de rendu de formulaire form.html.twig détaillé (1)

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

<h3>Formulaire d'annonce</h3>

<div class="well">
    {{ form_start(form, {'attr': {'class': 'form-horizontal'}}) }}

    {# Les erreurs générales du formulaire. #}
    {{ form_errors(form) }}

    {% if form.date is defined %}
        {# Génération du label + error + widget pour un champ. #}
        {# form_row(form.date) #}
        <div class="form-group">
            {{ form_label(form.date, "Date de l'annonce"
                        , {'label_attr': {'class': 'col-sm-2 control-label'}}) }}
            {{ form_errors(form.date) }}
            <div class="col-sm-10">
                {{ form_widget(form.date) }}
            </div>
        </div>
    {% endif %}
```

Symfony en pratique

Template de rendu de formulaire form.html.twig détaillé (2)

```
{# Génération manuelle et éclatée : #}
<div class="form-group">
    {# Génération du label. #}
    {{ form_label(form.title, "Titre de l'annonce"
                , {'label_attr': {'class': 'col-sm-2 control-label'}}) }}
    {# Affichage des erreurs pour ce champ précis. #}
    {{ form_errors(form.title) }}
    <div class="col-sm-10">
        {# Génération de l'input. #}
        {{ form_widget(form.title, {'attr': {'class': 'form-control'}}) }}
    </div>
</div>

{# form_row(form.content) #}
<div class="form-group">
    {{ form_label(form.content, "Contenu de l'annonce"
                , {'label_attr': {'class': 'col-sm-2 control-label'}}) }}
    {{ form_errors(form.content) }}
    <div class="col-sm-10">
        {{ form_widget(form.content, {'attr': {'class': 'form-control'}}) }}
    </div>
</div>
```

Symfony en pratique

Template de rendu de formulaire form.html.twig détaillé (3)

```
{# form_row(form.author) #}
<div class="form-group">
    {{ form_label(form.author, "Auteur de l'annonce"
                , {'label_attr': {'class': 'col-sm-2 control-label'}}) }}
    {{ form_errors(form.author) }}
    <div class="col-sm-10">
        {{ form_widget(form.author, {'attr': {'class': 'form-control'}}) }}
    </div>
</div>

{# form_row(form.published) #}

{# form_row(form.image) #}
<div class="form-group">
    {{ form_label(form.image, "Image de l'annonce"
                , {'label_attr': {'class': 'col-sm-2 control-label'}}) }}
    {{ form_errors(form.image) }}
    <div class="col-sm-10">
        {{ form_widget(form.image) }}
    </div>
</div>
```

Symfony en pratique

Template de rendu de formulaire form.html.twig détaillé (4)

```
    {# form_row(form.categories) #}
    <div class="form-group">
        {{ form_label(form.categories, "Catégories de l'annonce"
                    , {'label_attr': {'class': 'col-sm-2 control-label'}}) }}
        {{ form_errors(form.categories) }}
        <div class="col-sm-10">
            {{ form_widget(form.categories) }}
        </div>
    </div>
    {% if form.published is defined %}
        {{ form_row(form.published) }}
    {% endif %}
    <hr />
    {# Pour le bouton, pas de label ni d'erreur, on affiche juste le widget #}
    {{ form_widget(form.save, {'attr': {'class': 'btn btn-primary'}}) }}
    {# Génération automatique par Symfony des champs pas encore écrits.
       Par exemple, le champ CSRF (Cross Site Request Forgeries)
       et tous les champs cachés (type « hidden »). #}
    {{ form_rest(form) }}

    {# Fermeture de la balise <form> du formulaire HTML #}
    {{ form_end(form) }}
</div>
```

Symfony en pratique

Validation des données

- La validation des données va porter sur l'objet associé
- La configuration du mécanisme de validation peut être faite au niveau de l'entité auquel appartient l'objet
- Le déclenchement de la validation (dans le Contrôleur) est effectué par l'objet formulaire qui fait appel à la méthode isValid()

Symfony en pratique

Exemple de configuration du mécanisme de validation

```
use Symfony\Component\Validator\Constraints as Assert;
use Symfony\Component\Validator\ExecutionContextInterface;
use Symfony\Bridge\Doctrine\Validator\Constraints\UniqueEntity;
// …

/**
 * @var string
 *
 * @ORM\Column(name="title", type="string", length=255, nullable=false, unique=true)
 * @Assert\Length(min=10
 * , minMessage="Le titre doit faire au moins {{ limit }} caractères.")
 */
private $title;

/**
 * @var string
 *
 * @ORM\Column(name="author", type="string", length=255, nullable=false)
 * @Assert\Length(min=2
 * , minMessage="Le nom de l'auteur doit faire au moins {{ limit }} caractères.")
 */
private $author;
```

Symfony en pratique

La validation en général

- La validation est un principe à part entière et ne s'applique pas qu'aux formulaires
- Symfony est livré avec un composant Validator
- Il suffit pour chaque entité de configurer une liste de règles
  - Contraintes sur les nombres
  - Contraintes sur les chaines
  - Contraintes sur les dates
  - Contraintes sur les Collections
  - etc.

Symfony en pratique

La validation en général

- Les messages d'erreurs associés sont également paramétrables
- L'appel au composant Validator est implicite lors de la validation de formulaires
- Il peut être appelé si besoin dans le contrôleur :

Symfony en pratique

Appel du validateur dans un contrôleur

```
class AdvertController extends Controller {
// …
    public function testAction() {
        $advert = new Advert;
        $advert->setDate(new \Datetime());
        $advert->setTitle('abc');           // Champ « title » incorrect
        $advert->setAuthor('A');            // Champ « author » incorrect
        // On récupère le service validator
        $validator = $this->get('validator');
        // On déclenche la validation sur notre object
        $listErrors = $validator->validate($advert);

        if (count($listErrors) > 0) {
            // $listErrors est un objet,
            // La méthode __toString permet de lister joliement les erreurs
            return new Response((string) $listErrors);
        } else {
            return new Response("L'annonce est valide !");
        }
    }

}
```

Symfony
Notions avancées

Symfony, notions avancées

Zoom sur le routage

- Le framework offre beaucoup de possibilités

- Rappel : route permettant d'afficher notre liste d'articles

```
mysite_list_articles:
    pattern:  /test/list_articles
    defaults: { _controller: MySiteTestBundle:Article:list }
```

- Et si on a besoin de pagination pour afficher nos articles ?

```
mysite_list_articles:
    pattern:  /test/list_articles/{page}
    defaults: { _controller: MySiteTestBundle:Article:list, page:1 }
```

- On utilise un paramètre optionnel {page}
  - Optionnel car une valeur par défaut est fournie

Symfony, notions avancées

Zoom sur le routage

- Il est également possible de fournir des conditions
  - Le framework peut analyser certaines choses en plus du pattern pour savoir si la route correspond à l'URL entrée
- Condition sur la méthode HTTP utilisée

```
mysite_new_article_form:
    pattern:  /test/new_article
    defaults: { _controller: MySiteTestBundle:Article:newArticleForm }
    requirements:
        _method:  GET

mysite_new_article_submit:
    pattern:  /test/new_article
    defaults: { _controller: MySiteTestBundle:Article:newArticle }
    requirements:
        _method:  POST
```

- Condition sur un paramètre fourni (via expression régulière!)

```
mysite_list_articles:
    pattern:  /test/list_articles/{page}
    defaults: { _controller: MySiteTestBundle:Article:list, page:1 }
    requirements:
        page:  \d+
```

Symfony, notions avancées

Sécurité

- La sécurité dans une application est liée à deux concepts
  - L'authentification
  - Le contrôle d'accès (Habilitations)
- Le but est d'empêcher l'utilisateur d'accéder aux ressources auxquelles il n'a pas le droit d'accéder
- Symfony propose des outils pour faciliter ces 2 aspects

Symfony, notions avancées

Zoom sur la console

- Quelques commandes console utiles
  - Afficher toutes les routes de l'application
    - php bin/console debug:router
  - Afficher la liste des commandes consoles disponibles
    - php bin/console list
  - Vider le cache
    - php bin/console cache:clear
  - Afficher l'aide d'une commande
    - php bin/console help maCommande

Symfony, notions avancées

Les Front Controllers

- Symfony propose 2 contrôleurs frontaux

  On utilise qu'un seul Front Controller à la fois

  - Un pour le développement
    - Au niveau du projet : `web/app_dev.php`

  - Un pour la production
    - Au niveau du projet : `web/app.php`

Symfony, notions avancées

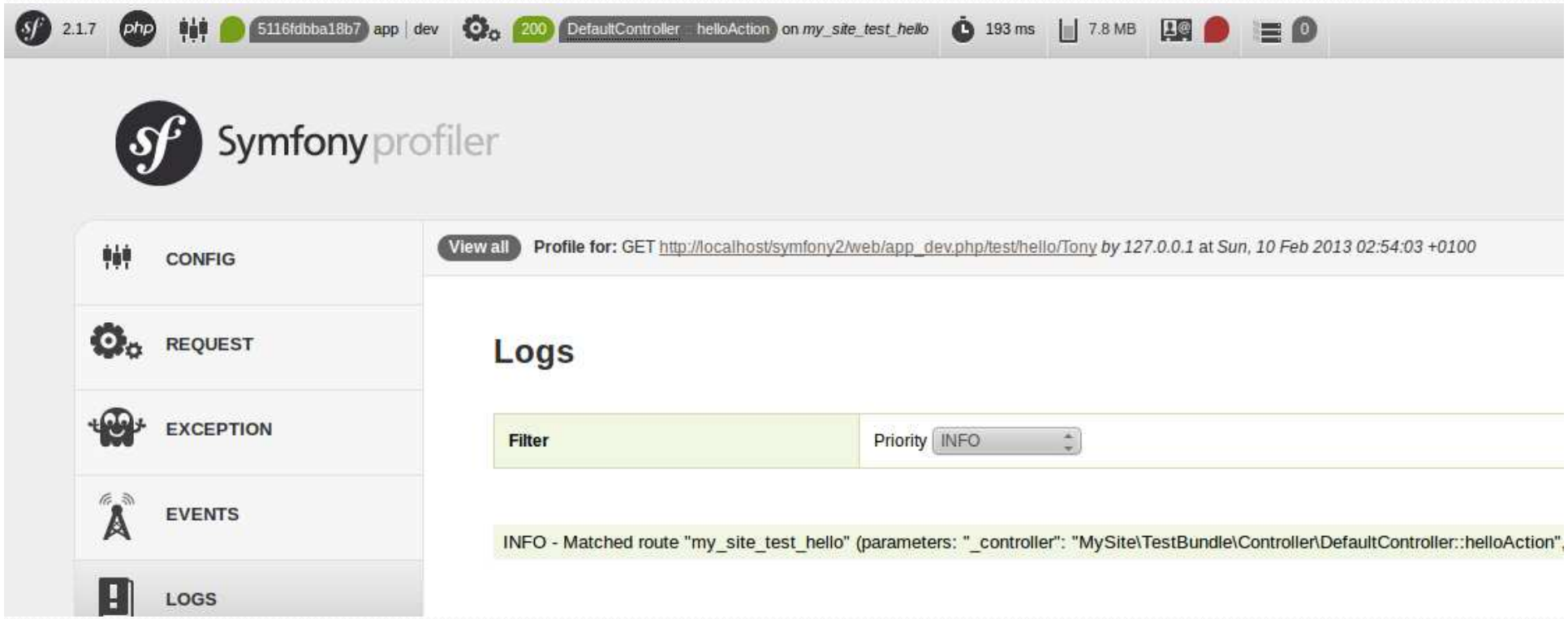
Les Front Controllers : Contrôleur pour le dev

- Le contrôleur « `app_dev.php` » offre toute une batterie d'outils de debug pour le développeur
  - Logs des informations présentes dans la requête
  - Informations détaillées sur le fonctionnement du framework
  - Affichage détaillé en cas d'exception
  - Temps d'exécutions des scripts
  - Cache du framework désactivé
  - Consultation des routes déclarées

Symfony, notions avancées

Les Front Controllers : contrôleur pour le dev

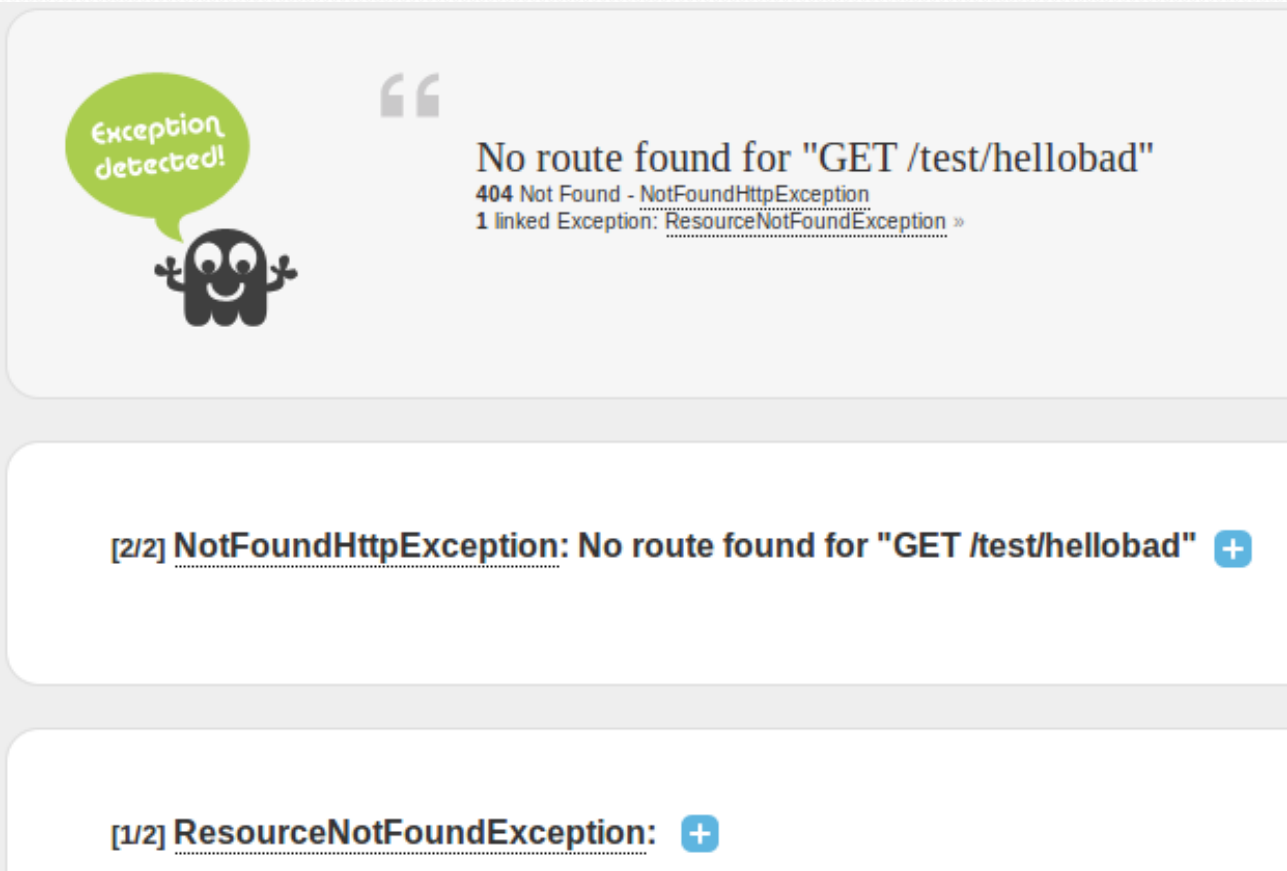
- Exemple : barre de debug symfony



Symfony, notions avancées

Les Front Controllers : contrôleur pour le dev

- Exemple d'exception en environnement de dev



Symfony, notions avancées

Les Front Controllers : contrôleur de production

- Le contrôleur « `app.php` »
  - Adapté à la production,
  - Cache activé
  - Erreurs non détaillées, adaptées à l'utilisateur

Symfony, notions avancées

Les Front Controllers : contrôleur de production

- Page d'erreur adaptée à la production

Oops! An Error Occurred

The server returned a "404 Not Found".

Something is broken. Please e-mail us at [email] and let us know what you were doing when this error occurred.

- Une ressource n'a pas été trouvée, donc on affiche une erreur 404

Symfony, notions avancées

Configuration et paramètres

- Offre un mécanisme de configuration pour l'application
  - Déclaration de paramètres
    - app/config/parameters.yml

```
parameters:
    test_param:        MyTest
```

  - Utilisation d'un paramètre dans un contrôleur

```
$this->container->getParameter('test_param');
```

Beaucoup plus simple que de lire un fichier !

Bibliographie

- Site officiel de Symfony
  - *https://www.symfony.com/*
    - *Quick Start Guide*
    - *Book*
    - *CookBook*

- *Tutoriels pour Symfony*
  - *https://openclassrooms.com/courses/**developpez-votre-site-web-avec-le-framework**-symfony*
  - *http://jobeet.thuau.fr/*

Conclusion

- Boite à outils très complète
- Framework léger et très modulaire
- Moins de code de « plomberie » à écrire pour le développeur. On se concentre sur le code « métier »
- On ne réinvente pas la roue !
- Demande un certain temps d'adaptation – et des connaissances de base solides
- Documentation complète