

---


R4.01

# Architecture Logicielle

---

**Introduction**

# Objectifs

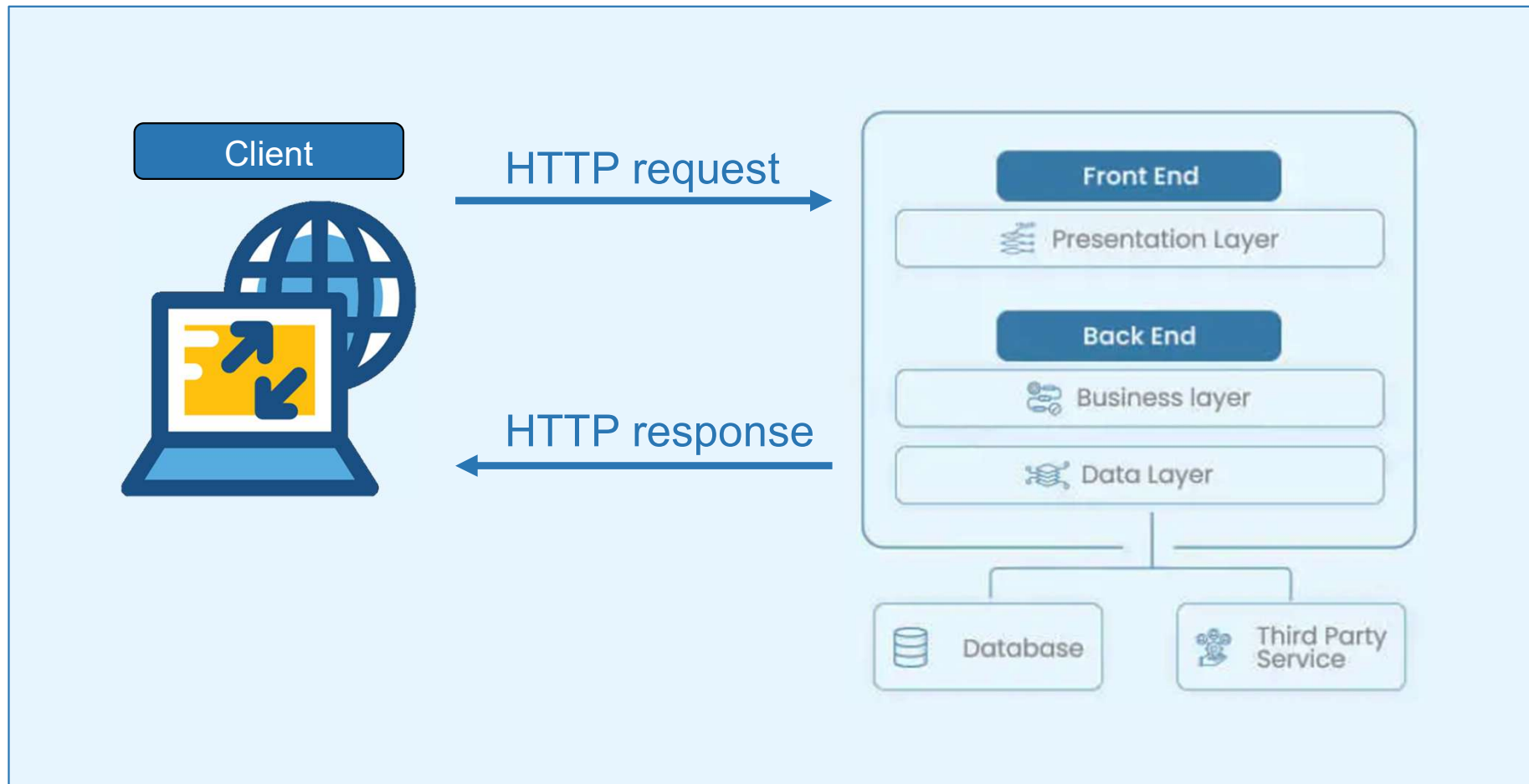
- Développer une application Web correctement structurée
- Etudier les méthodes et les outils permettant de le faire : framework MVC, moteur de template, orm, ...
- Travailler en techno PHP avec  Symfony

# Application Web ?



- En informatique, une **application web** (aussi appelée **web application**, de l'anglais) est une **application** manipulable grâce à un **navigateur web**. De la même manière que les sites web, une application web est généralement placée sur un serveur et se manipule en actionnant des *widgets* à l'aide d'un navigateur web, via un réseau informatique (Internet, intranet, réseau local, etc.).
- Des messageries web, les systèmes de gestion de contenu, les wikis et les blogs sont des applications web.
- Les moteurs de recherches, les logiciels de commerce électronique, les jeux en ligne, les logiciels de forum peuvent être sous forme d'application web.
- Les applications web font partie de l'évolution des usages et de la technologie du Web appelée Web 2.0.

# Architecture de Base



Adapté de Moontechnolabs

# La Mauvaise Approche pour Développer

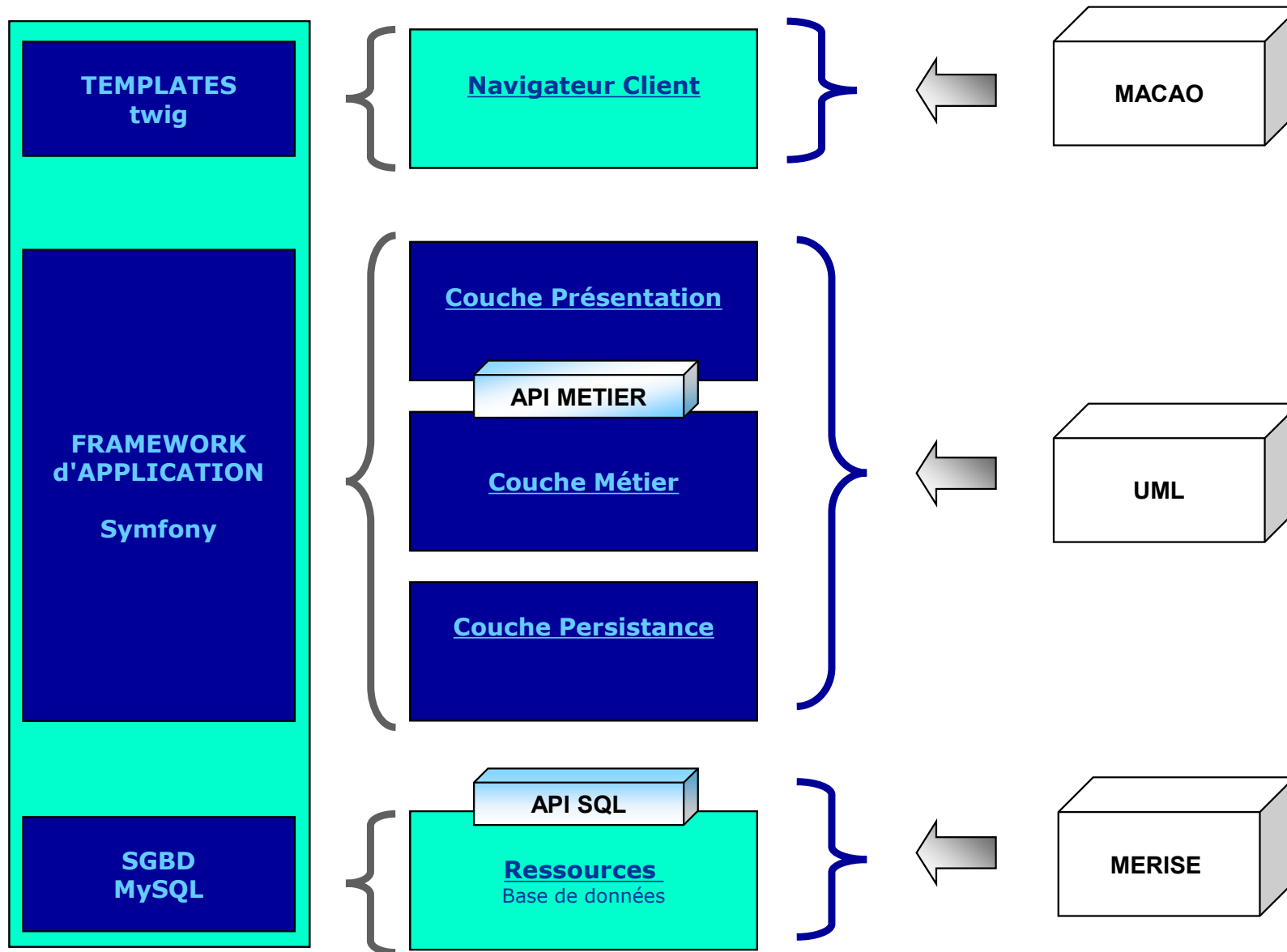
- L'approche "*Personnal Home Page*" ou "Programmation à Plat" : on considère l'application comme une collection de pages Web
- Chaque page est un script PHP, plus ou moins compliqué, qui traite une requête, accède à une BD, réalise un traitement et affiche un résultat sous forme HTML
- Résultat : tous les niveaux logiques sont mélangés et dès que l'application devient suffisamment importante, elle n'est plus maintenable.

# Bonne approche : dissocier les niveaux

- Niveau présentation
  - Navigateur / Serveur web
- Niveau applicatif
  - Script ou programme
- Niveau données
  - Accès aux données

Résultat : une application structurée, avec des composants indépendants ayant un faible niveau de couplage, facile à développer et à maintenir

# Architecture en couches

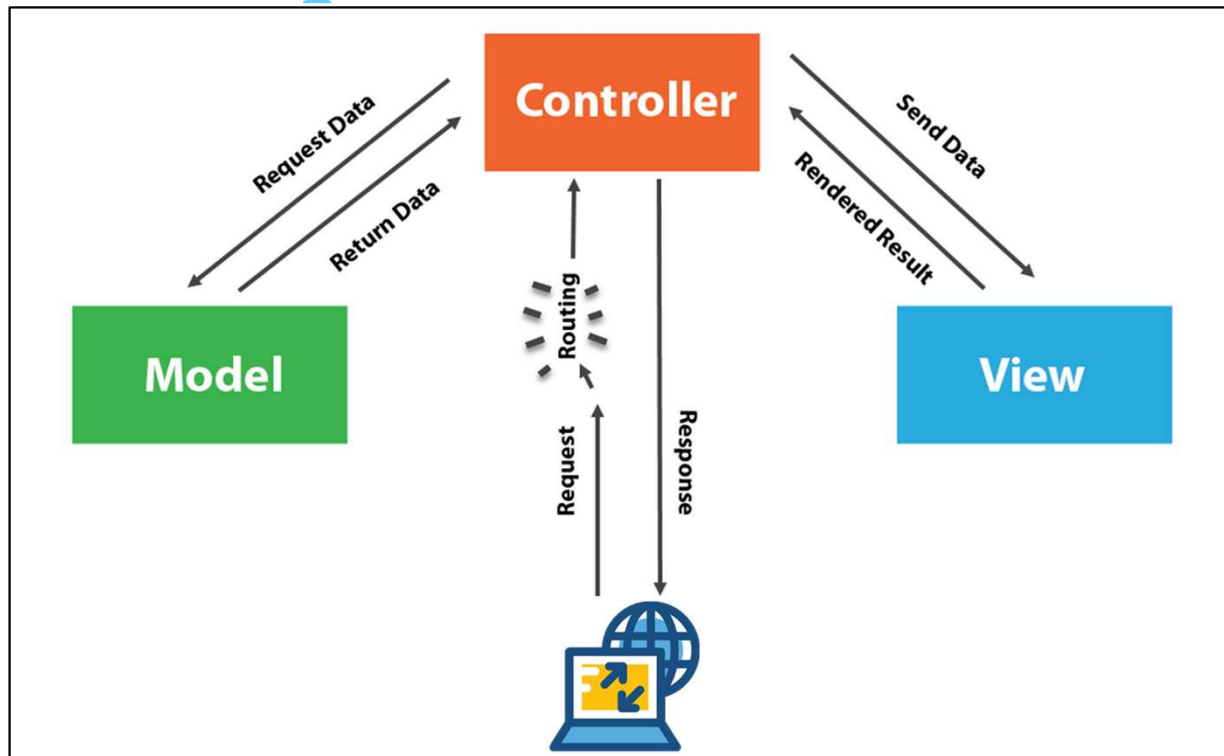


# La couche client léger

- C'est le client (navigateur Web) qui est chargé de l'affichage
- HTML + JavaScript + css
- Eventuellement, elle réalise un certain nombre de contrôles de premier niveau pour faciliter la vie de l'utilisateur (et non pour assurer la sécurité de l'application)



# La couche presentation : le C du MVC



Adapté de [ozzu.com](http://ozzu.com)

- MVC : Architecture utilisée pour concevoir et implanter des Interfaces Homme-Machine (IHM)
  - Inventée à Xerox Parc dans les années 1970
  - Première apparition "publique" dans SmallTalk'80
  - Toutes les architectures actuelles pour les applications interactives s'inspirent de ce modèle.
  - MVC et son évolution MVC2 sont supportés par de nombreux frameworks de développement Web
    - en Java : Struts, Barracuda, Hammock, ...
    - en PHP : CakePHP, Zend Framework, Code Igniter, Symfony, Laravel...
    - En ASP : ASP.NET MVC

# La Couche Métier



- Cœur de l'application
- Définit l'API métier de l'application
- On y Accède depuis la couche présentation
  - pas d'accès remontant vers la couche présentation
- Accède à la couche persistance
  - pas d'accès direct à la base de données

# La couche persistance

- Réalise une interface orientée objet pour manipuler les données contenues dans la BD
- C'est donc la seule couche qui dialogue directement avec la BD



# Framework

- Les **frameworks** sont des structures logicielles qui définissent des cadres dans lesquels viennent s'insérer les objets et concepts spécifiques à une application.
- En pratique, un **framework** (squelette d'application) est un ensemble de classes et de mécanismes associés à une **architecture logicielle** (par exemple MVC) qui fournissent un ou plusieurs services techniques (ou métiers) aux applications qui s'appuient dessus.

# Framework

- Un **framework métier** fournit des services à forte plus-value fonctionnelle (gestion de clients, d'abonnements, de news, ...)
- Un **framework technique** apporte les concepts, entités et mécanismes qui permettent, dans le cadre d'une architecture logicielle retenue, de s'abstraire d'un certain nombre de problématiques conceptuelles et techniques récurrentes.
- Par exemple, **Symfony** est un **framework technique** PHP pour les développements Web.



- Les Frameworks ont l'avantage de structurer, simplifier, segmenter les développement et donc de les accélérer (on parle de framework de développement rapide)
- Un **framework technique** peut prendre en compte les problématiques de performance, sécurité, multi-langue, ...

# Intérêts d'un développement WEB basé sur un framework MVC 2

- Les applications conçues ainsi sont plus facilement maintenables et évolutives, du fait de la séparation des fonctionnalités du modèle, du contrôle, et de la vue.
- L'utilisation du framework permet au développeur de se concentrer sur le **modèle**, le framework fournissant le cadre nécessaire au contrôle et à l'interface de l'application, permettant un développement rapide de ces parties.
- **Attention** : l'emploi d'un framework nécessite un investissement : un certain temps est nécessaire pour prendre en main le concept et les API.
- La centralisation des accès à l'application permet un contrôle fin et personnalisé des accès aux traitements (Actions) et offre une grande marge de manœuvre pour la gestion des profils ou des rôles utilisateurs.
- Les sources de nombreux frameworks sont publiques. **Ils sont donc aisément adaptables et extensibles.**
- ...

---

R4.01

# Applications Web



# Pourquoi Symfony ?

Si vous savez développer en PHP, pas besoin d'un framework... Mais c'est beaucoup mieux d'en avoir un :

## ■ Pour simplifier et accélérer le développement

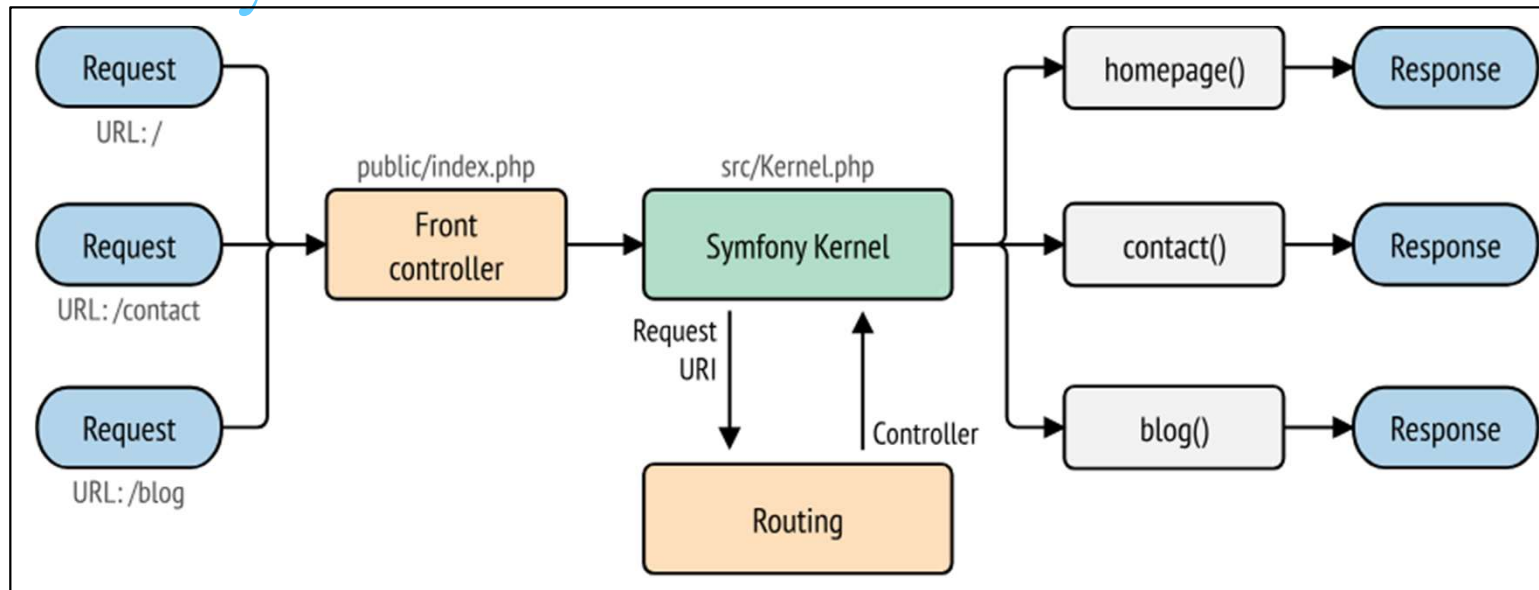
- Travailler mieux : structurer le développement
- Travailler plus vite en réutilisant des modules génériques
- Faciliter la maintenance à long terme et l'évolution en se conformant à des règles de développement standards
- Simplifier l'intégration et l'interfaçage d'une application avec le reste du système d'information

## ■ Pourquoi Symfony?

- Environnement de développement utilisé dans l'industrie, reconnu, stable,
- Documentation importante
- Communauté active
- Ouvert : intègre des composants qui peuvent être utilisés indépendamment du framework



# Symfony : framework HTTP ? MVC ?

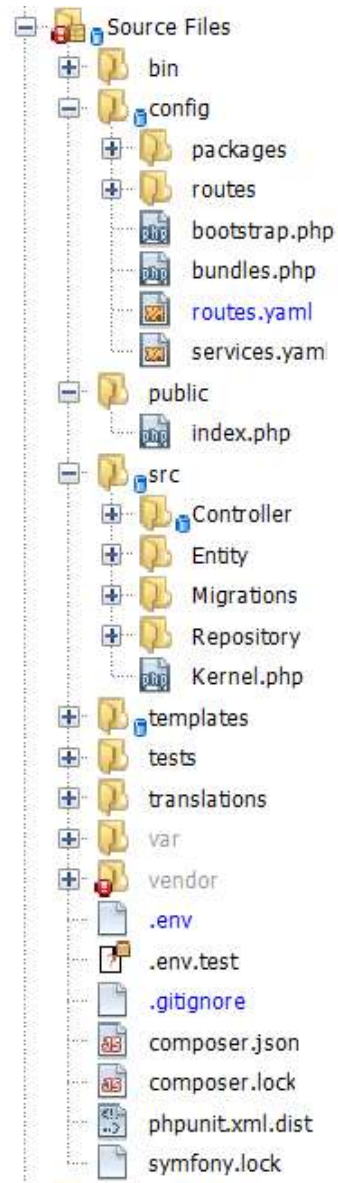


- Chaque requête qui peut être soumise à l'application est définie sous forme d'attributs dans le code des routeurs PHP (ou éventuellement sous forme de fichier xml ou yaml)
- Un **route** définit une correspondance entre une URL et la méthode PHP (**Action**) qui sera exécutée lorsque cette URL est soumise à l'application.
- Le **Contrôleur frontal** (index.php) analyse l'URL, cherche la route correspondante et exécute l'action correspondante
- La tâche de chaque **action** est de traiter la requête HTTP (objet **Request**) reçue puis de renvoyer une réponse HTTP (objet **Response**).
- Les actions (et donc les URLs) qui relèvent d'un même domaine fonctionnel sont développées au sein d'une même classe appelée **Contrôleur**
- Dans une application Web « traditionnelle », la réponse consiste en une **vue** fabriquée à partir d'un **template Twig** auquel l'**action** aura transmis des informations issues du **Modèle**

# Structure d'une application (1)

- Le code d'une application développée en symfony est développé dans un **bundle** par défaut appelé **App**
- Symfony lui-même est une collection de bundles
- Un bundle est un ensemble structuré de fichiers au sein d'un répertoire et qui implémentent une fonctionnalité unique.
- Chaque répertoire contient tout ce qui est lié à cette fonctionnalité incluant les fichiers PHP (contrôleurs, objets métiers), les templates, les feuilles de style, le javascript, les tests et tout le reste...
- Depuis Symfony 4, il n'est plus préconisé de créer plusieurs Bundles pour structurer son application.  
Il est plutôt conseillé d'utiliser la notion PHP de « NameSpace »

# Structure d'une application (2)



- **bin** : interface CLI
- **config** : configuration du projet
- **public** : répertoire visible par le serveur Web : le contrôleur frontal
- **src** : code source du projet
  - Controller : les contrôleurs que vous allez développer
  - Entity : les entités (objets métier) => interfacent des n-uplets de la BD
  - Repository : les dépôts d'objets métier => interfacent les tables de la BD
- **templates** : les « vues » du projet (templates Twig)
- **tests** : les tests unitaires, fonctionnels développés pour le projet
- **translations** : les dictionnaires pour l'internationalisation du projet
- **.env** : les variables d'environnement du projet
- **composer.json** : le fichier de configuration des composants du projet

# Routing

- Les routes peuvent être définies de plusieurs façons (au choix !) :
  - Soit par un ou plusieurs fichiers de configuration au format YAML

Ce fichier définit 2 routes (alignées à gauche)

Les propriétés de toutes les entrées doivent être décalées avec le même nombre d'espaces

```
# config/routes.yaml
api_post_show:
  path:      /api/posts/{id}
  controller: App\Controller\BlogApiController::show
  methods:   GET|HEAD
api_post_edit:
  path:      /api/posts/{id}
  controller: App\Controller\BlogApiController::edit
  methods:   PUT
```

Pas utilisé en TP !

- Soit dans le code PHP des contrôleurs à l'aide d'attributs PHP : **`#[Route(...)]`**

```
use Symfony\Component\Routing\Annotation\Route;

class DefaultController extends AbstractController {

    #[Route('/default', name: 'app_default')]
    public function index() : Response {
        // ...
    }
}
```

Utilisé en TP !

# Routing

- Une application typique comporte de nombreuses routes.
- Une route basique a une URL (path), un nom (name) et éventuellement plusieurs attributs
  - Le **path** représente l'URL qui est définie par cette route, par exemple **/default**
  - Le **nom** est un identifiant libre et unique, par exemple **app-default**
  - La méthode exécutée lorsque cette route sera utilisée est celle devant laquelle l'attribut **#[Route(...)]** est défini

*Attributs PHP*

```
use Symfony\Component\Routing\Annotation\Route;

class DefaultController extends AbstractController {

    #[Route('/default', name: 'app_default')]
    public function index(): Response {
        // ...
    }
}
```

# Routing

- Une route peut contenir un ou plusieurs paramètres de substitution nommés « joker »
- Le path va faire correspondre tout ce qui ressemble à /blog/.\*
- La valeur correspondant au paramètre de substitution **{slug}** sera disponible dans votre contrôleur.
- => si l'URL est /blog/hello-world, un paramètre **\$slug**, valant "hello-world", sera transmis au contrôleur

*Attribut PHP*

```
use Symfony\Component\Routing\Annotation\Route;

class BlogController extends AbstractController {

    #[Route('/blog/{slug}', name: 'blog_show')]
    public function show(string $slug): Response {
        // ...
    }
}
```

# De la route à la vue en passant par le contrôleur

- On définit une route **hello\_page** par annotation dans son contrôleur:
- Toute URL de la forme **/hello/\*** exécutera donc l'action **hello** du **DefaultController** :

```
// Controller/DefaultController.php
namespace App\Controller;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Routing\Attribute\Route;

class DefaultController extends AbstractController {

    #[Route('/hello/{userName}', name: 'hello_page',
           defaults=["username" => "Anonyme"])]

    public function hello(string $userName) : Response {
        return $this->render("hello.html.twig", ["userName" => $userName,]);
    }
}
```

- Et l'exécution de cette action affichera la vue **hello.html.twig** suivante :

```
{# templates/hello.html.twig #}
<!-- mettre tout le code HTML nécessaire autour...-->
Hello {{ userName }} !
<!-- mettre tout le code HTML nécessaire autour...-->
```

---

# Moteur de Templates Twig

---

*Le strict minimum à savoir...*

<https://twig.symfony.com/>

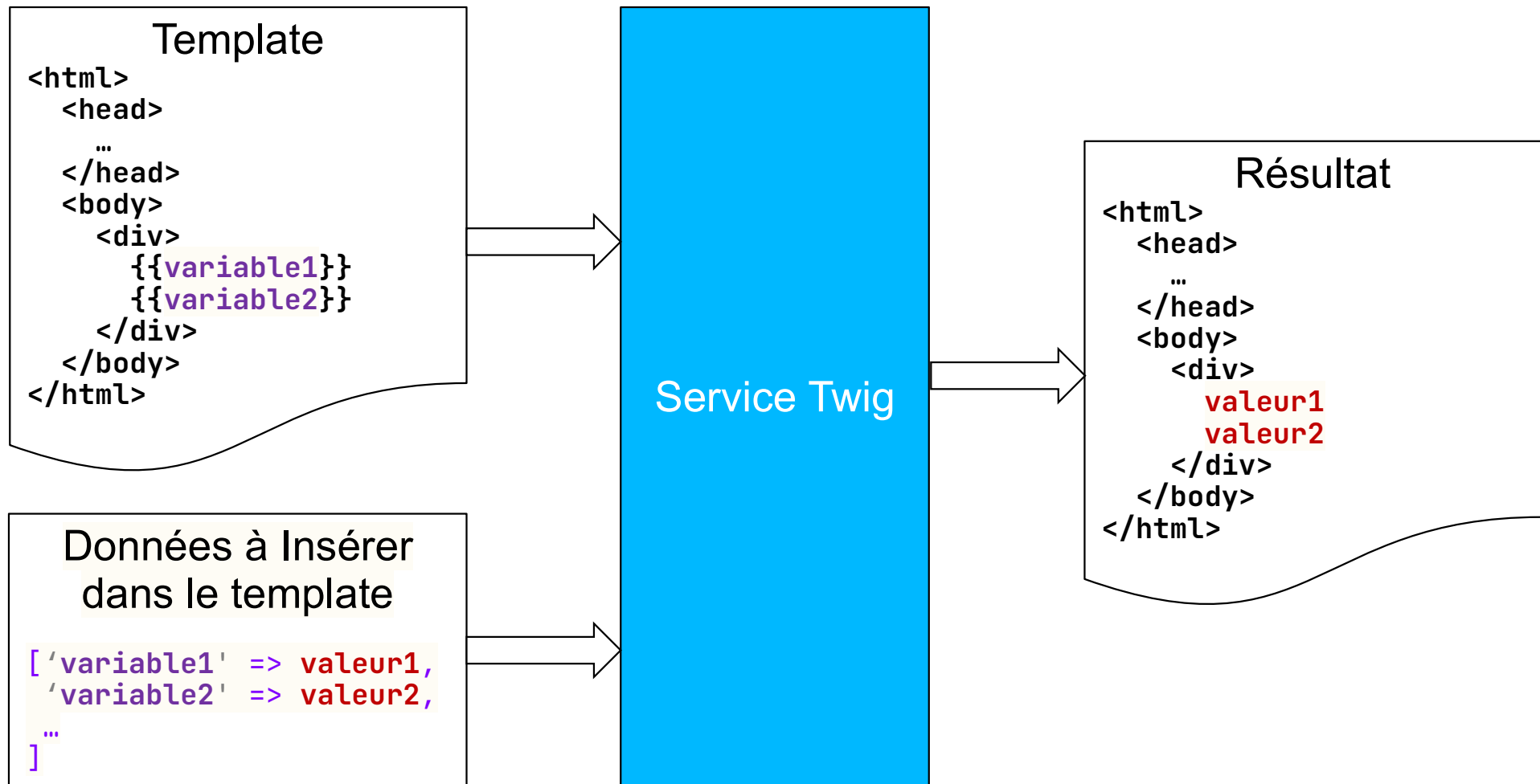


# Les templates (vues) : Twig

- Un template Twig est un fichier contenant des balises Twig qui seront interprétées par le moteur de rendu
- Le template peut être de n'importe quel format texte : HTML, XML, PDF, ...
- Les balises Twig peuvent être insérées n'importe où dans le texte

# Le service Twig

- Le moteur de template Twig est un **service** de symfony : un objet PHP, qui propose des méthodes pour réaliser une certaine fonctionnalité et qui peut être injecté automatiquement là où il est utilisé



# Les Balises Twig

- Twig définit trois sortes de balises :
  - ❑ `{{ ... }}`: affiche une variable ou le résultat d'une expression.
  - ❑ `{% ... %}`: Contrôle la logique du *template*; utilisé pour exécuter des boucles **for** et des conditions **if**, entre autres.
  - ❑ `{# ... #}` : commentaires

# Exemple (1)

- Un contrôleur transmet au template **exemple.html.twig** deux variables :
  - Une chaîne (**titre**) contenant le titre de la page
  - Un tableau de tableaux associatif (**unTableau**) représentant un ensemble d'URLs avec un texte associé à chaque URL :

```
// src/Controller/DefaultController.php

namespace App\Controller;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class DefaultController extends AbstractController {

    public function exemple() : Response {
        return $this->render('Default/exemple.html.twig', [
            "titre" => "Page d'Exemple",
            "unTableau" => [
                ["url" => "http://url1.com", "texte" => "Lien vers l'URL 1"],
                ["url" => "http://url2.com", "texte" => "Lien vers l'URL 2"],
            ]
        ]);
    }
}
```

# Exemple (2)

## ■ Le template `exemple.html.twig` :

```
{# templates/Default/exemple.html.twig #}
<!DOCTYPE html>
<html>
  <head>
    <title>Une Page</title>
  </head>
  <body>
    <h1>
      {% if titre is defined %}
        {{ titre | upper }} {# upper est un filtre twig #}
      {% else %}
        Titre par Défaut
      {% endif %}
    </h1>
    <ul id="navigation">
      {% for item in unTableau %}                                     {# item est un élément de unTableau
                                                                                   c'est donc un tableau associatif #}
        <li>
          <a href="{{ item.url }}">
            {{ item.texte }}
          </a>
        </li>
      {% endfor %}
    </ul>
  </body>
</html>
```

# Exemple(3)

- Le code HTML rendu par le moteur Twig :

```
<!DOCTYPE html>
<html>
  <head>
    <title>Une page</title>
  </head>
  <body>
    <h1>
      Page d'Exemple
    </h1>
    <ul id="navigation">
      <li><a href="http://url1.com">Lien vers l'URL 1</a></li>
      <li><a href="http://url2.com">Lien vers l'URL 2</a></li>
    </ul>
  </body>
</html>
```

- Les données transmises par le contrôleur et insérées dans le *template* sont automatiquement « échappées » pour éviter les failles de sécurité de type XSS
- Utiliser le filtre `raw` pour que l'échappement ne soit pas fait

# Les templates (vues) : Twig

- Les variables passées à un *template* peuvent être de tous types : nombres, chaînes de caractères, tableaux ou objets PHP.
- Twig gère les tableaux associatifs et les objets de manière intuitive et permet d'accéder aux « attributs » d'une variable grâce à la notation pointée (.)

```
{# array('name' => 'Fabien') #}  
{{ name }}  
  
{# array('user' => array('name' => 'Fabien')) #}  
{{ user.name }}  
  
{# accès explicite à un tableau associatif #}  
{{ user['name'] }}  
  
{# array('user' => new User('Fabien')) #}  
{{ user.name }}  
{{ user.getName }}  
  
{# appel d'une méthode sur un objet user #}  
{{ user.name() }}  
{{ user.getName() }}  
  
{# transmission de paramètres à une méthode #}  
{{ user.date('Y-m-d') }}
```

# Les templates (vues) : Twig - Héritage

- Un *template* peut être « décoré » par un autre par un mécanisme d'héritage.
- L'héritage de *template* permet de bâtir un *template* « layout » de base qui contient tous les éléments communs de votre site et de définir des « blocks » que les *templates* enfants pourront surcharger.
- Le tag `{% block %}` définit des blocs que les *templates* enfants vont pouvoir **redéfinir**
  - Exemple : définition dans le fichier `base.html.twig` d'un block appelé « **body** »

```
{# templates/base.html.twig #}  
<div class="symfony-content">  
    {% block body %}  
    {% endblock %}  
</div>
```

- Le *template* `hello.html.twig` hérite du *template* `base.html.twig` et **redéfinit** le contenu du block « **body** »

```
{# templates/Demo/hello.html.twig #}  
{% extends "base.html.twig" %}  
  
{% block body %}  
    <h1>Hello {{ name }}!</h1>  
{% endblock %}
```



# Les templates (vues) : Twig - Inclusion

- Pour partager un morceau de vue (un fragment) entre plusieurs *templates* distincts, on peut créer un nouveau *template* qui sera inclus dans les autres.
- Exemple : le template **embedded.html.twig** pourrait être un fragment qui affiche « hello » suivi du nom de l'utilisateur transmis dans une variable « name » :

```
{# templates/Demo/embedded.html.twig #}  
Hello {{ name }}
```

- Et le template **hello.html.twig** pourrait inclure ce fragment (ici dans le block « content ») :

```
{# template/demo/hello.html.twig #}  
  
{% extends "base.html.twig" %}  
  
{% block body %}  
    {% include "demo/embedded.html.twig" %}  
{% endblock %}
```

# Les templates (vues) : Twig - Liens

- Pour garder le bénéfice du *routing*, il ne faut pas coder « en dur » les URLs dans les *templates*
- La fonction **path** génère des **URLs relatives** en se basant sur la configuration du *routing*.
- Ainsi toutes vos URLs peuvent être facilement mises à jour en changeant juste le fichier de configuration

**demo\_hello** est le nom d'une route qui a un paramètre appelé **name**

```
<a href="{{ path('demo_hello', { 'name': 'Fabien' }) }}">Hello Fabien!</a>
```

- Quand une route attend des paramètres, il faut les lui transmettre dans un tableau associatif au format « twig ».  
Exemple : `{ 'name' : 'Fabien' }`

- La fonction **url** génère des **URLs absolues** à partir d'une route:

```
<a href="{{ url('demo_hello', { 'name': 'Thomas' }) }}">Hello Thomas !</a>
```

# Les templates (vues) : Twig - Assets

- Pour utiliser des ressources (images, css, ...) dans un template Twig de façon « portable », il faut utiliser la fonction **asset**
- Grâce à cette fonction, vous pouvez déplacer le répertoire de votre application sans devoir changer le code de vos *templates*.

```
<link href="{{ asset('css/blog.css') }}"  
      rel="stylesheet" type="text/css" />  
  

```

- Les ressources (images, css, ...) doivent être placées dans le répertoire **public** de votre projet
- Le chemin passé en paramètre à **asset** est un chemin **relatif** par rapport au répertoire **public**

# Pour plus de détails sur Twig...

- Doc Symfony :

- <https://symfony.com/doc/current/templates.html>

- Doc Twig

- <https://twig.symfony.com/>