# Présentation
# Architecture MVC en PHP avec Silex

Antoine JAMIN

Août 2016

---

**Langages et notions abordés :** MVC, PHP, Silex, HTML, Twig et Bootstrap.

MVC est une architecture qui permet de différencier les trois parties suivantes :

- Modèle
- Vue
- Contrôleur

Cette présentation permet d'avoir un premier aperçu avant de s'attaquer à la pratique.

Ce tutoriel s'appui sur les travaux suivants :

- `https://openclassrooms.com/courses/evoluez-vers-une-architecture-php-professionnelle`
- `http://silex.sensiolabs.org/`
- `http://twig.sensiolabs.org/`

Dans cette présentation nous essaierons dans un premier temps de définir le MVC, puis dans une deuxième partie nous détaillerons tous les éléments pour la mise en place d'une telle architecture en PHP.

## 1 MVC

Cette architecture permet d'avoir une approche structurée d'un projet avec une architecture beaucoup plus claire. Elle permet également d'obtenir trois niveaux complètement autonomes. Si par exemple vous souhaitez changer de type de base de données, il vous suffira de modifier uniquement la couche modèle. Elle apporte donc modularité et facilité de maintenance.

### 1.1 Modèle

Dans la description d'une architecture en couches, la partie modèle s'apparente à la couche métier, elle permet d'organiser les données et de les rendre utilisables par le contrôleur. Elle peut se résumer à une partie algorithmique où l'on déclare les différentes classes et leurs fonctions.

### 1.2 Vue

Dans la description d'une architecture en couches, la partie vue s'apparente à la couche interface. Elle gère les différents affichages de chaque partie de l'application. Elle assure le lien avec l'utilisateur car c'est à elle qu'il est confronté.

### 1.3 Contrôleur

Permet de mettre en lien les deux précédentes parties, elle gère les évènements utilisateurs et leur apporte la réponse nécessaire. Si par exemple l'utilisateur clique sur le bouton pour afficher les données de la base de données, le contrôleur utilisera la fonction d'affichage définie dans le modèle et affichera son résultat à l'aide de la partie vue.

## 2 Mise en place de l'architecture

Pour illustrer notre exemple nous utiliserons le Framework PHP Silex qui est un Framework simplifié de Symfony.

### 2.1 Arborescence du projet

Un projet Silex s'organise selon l'arborescence suivante :

```
/
├── app
│   ├── config
│   ├── app.php
│   └── routes.php
├── src
│   ├── DAO
│   └── Domain
├── vendor
├── views
├── web
│   ├── css
│   ├── img
│   ├── js
│   ├── lib
│   ├── index.php
│   └── .htaccess
└── composer.json
```

### 2.2 app

C'est le dossier permettant de gérer la partie contrôleur. Il est composé du fichier `app.php` qui définit les propriétés de l'application et de ses composantes dont les configurations sont renseignées dans le dossier config. Il est également composé d'un fichier `routes.php` qui permet de définir tous les chemins de l'application. Par exemple que fait l'application lorsque le chemin est '/' (correspondant au premier chemin d'une application).
Illustrons cette partie à l'aide du code suivant :

```php
<?php

//Declaration des dependances PHP
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

//Route pour la page principale
$app->get('/', function() use ($app){
  //declarer ce que fait cette page
  $i=0; //...
  //redirige vers le chemin Metier
  return $app->redirect($app['url_generator']->generate('Metier'));
```

```
})->bind('home');

//Route pour la page metier
$app->get('/Metier/', function() use ($app){
  //declarer ce que fait cette page
  $metiers = $app['dao.metier']->findAll(); //...
  //Fait appel a la vue metier.html.twig
  return $app['twig']->render('Metier.html.twig', array('metiers'=>$metiers)
   );
})->bind('Metier')
```

routes.php

### 2.3 src

C'est le dossier permettant de gérer la partie modèle. Un dossier `DAO` permet de gérer les interactions avec la base de données et un dossier `Domain` permet de gérer les différentes classes. Souvent il y a un fichier DAO pour chaque classe du Domain. Par exemple si nous déclarons la classe `Métier` nous crérons aussi une classe `MétierDAO`. La classe `Métier` contient toutes les déclarations des variables avec leurs accesseurs et leurs mutateurs. La classe `MétierDAO` contiendra toutes les méthodes pour consulter les différents métiers de la base.
Illustrons cette partie par une classe métier ayant pour attribut un identifiant (id) et un nom. Le code correspondant serait le suivant :

```
<?php

namespace Application\Domain;

class Metier
{
  //declarations des variables
  private $id;
  private $nom;

  //declarations des accesseurs (get) et mutateurs (set)
  public function getId(){
    return $this->id;
  }
  public function setId($id){
    $this->id=$id;
  }

  public function getNom(){
    return $this->nom;
  }
  public function setNom($nom){
    $this->nom=$nom;
  }
}
```

Metier.php

```
<?php

namespace Application\DAO;

class JobDAO extends DAO
{
  public function findAll(){
```

```
8       $sql='SELECT * FROM metier ORDER BY nom';
9       $result=$this->getDb()->fetchAll($sql);
10      $metiers=array();
11      foreach($result as $row){
12        $id=$row['id'];
13        $metiers[$id]=$this->buildDomainObject($row);
14      }
15      return $metiers;
16    }
17
18    protected function buildDomainObject($row){
19      $metier = new Metier();
20      $metier->setId($row['id']);
21      $metier->setNom($row['nom']);
22      return $metier;
23    }
24  }
```

MetierDAO.php

### 2.4 vendor

Ce dossier est généré automatiquement par `composer` un outil pour le management des dépendances PHP (`https://getcomposer.org/`).
Toutes les dépendances utilisées dans le projet sont déclarées dans le fichier `composer.json`.

### 2.5 views

C'est le dossier permettant de gérer la partie vue.
Le fichier pour l'affichage peut être en plusieurs formats. Dans un but de simplicité nous avons choisis de travailler avec Twig, un moteur de template permettant à la fois de gérer des variables et aussi du code HTML.
Chaque type de vue est défini dans un fichier `.html.twig` et chacun peut faire appel à une template à l'aide de la fonction {% `extends` %}.

Illustrons cette partie par le code suivant :

```
1  <!doctype html>
2  <html>
3  <head>
4      <meta charset="utf-8">
5      <meta http-equiv="X-UA-Compatible" content="IE=edge">
6      <meta name="viewport" content="width=device-width, initial-scale=1">
7      <link href="{{ app.request.basepath }}/lib/bootstrap/css/bootstrap.min.
       css" rel="stylesheet">
8      <title>{% block title %}{% endblock %}</title>
9      <link rel="icon" type="image/png" href="{{ app.request.basepath }}/img/
       logo.png">
10     <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.
       min.js"></script>
11     <script src="http://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap
       .min.js"></script>
12     <link href="{{ app.request.basepath }}/css/style.css" rel="stylesheet">
13 </head>
14 <body>
15
16 {% block content %}{% endblock %}
17
```

```
<script src="{{ app.request.basepath }}/js/functions.js" type="text/
    javascript"></script>
</body>
</html>
```

layout.html.twig

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

{% block titre %} éMtiers {% endblock %}

{% block corps %}
<h1>Liste des éMtiers :</h1>
<table>
  <thead><tr><th>Index</th><th>Nom</th></tr></thead>
  <tbody>
    {% for metier in metiers %}
      <tr><td>{{ metier.id }}</td><td>{{ metier.nom }}</td></tr>
    {% endfor %}
  </tbody>
</table>
{% endblock %}
```

metier.html.twig

### 2.6 web

C'est le dossier qui est le cœur du projet web.
C'est vers lui que doit pointer le serveur pour pouvoir afficher cette application.
Il contient le fameux `index.php` qui permet de mettre en run l'application et d'appeler toute l'architecture.
Il contient aussi tous les dossiers contenant le CSS, les images (img), le code javascript (js) et les différentes librairies (lib). Ces dossiers pourront être appelés dans le code html de la template par exemple.