

CAHIER TECHNO

FRAMEWORK PHP

Un test « en temps réel » sur un
cas d'école



TheCodingMachine™
TCM://

Introduction

Qu'y a-t-il dans ce cahier techno ?

Nous proposons une approche plutôt pragmatique pour présenter les principaux frameworks PHP. Nous nous sommes dit que les principaux critères de choix d'un framework étaient d'en comprendre facilement le fonctionnement, réussir à le prendre en main rapidement et avoir accès à une documentation complète. Fort de ce constat, nous avons choisi de tester (en essayant d'être le plus objectif possible) la mise en œuvre d'un cas d'école.

En bref, ce cahier a pour but de dépasser la discussion d'experts qui ont la fâcheuse tendance à finir par préconiser ce qu'ils connaissent !

La première partie fait une rapide synthèse des raisons qui poussent à utiliser un framework de manière générale et dans la deuxième partie, David Négrier, notre CTO réalise et commente en « temps réel » le développement de ce test.

A qui s'adresse ce livre blanc ?

Evidemment, ce cahier s'adresse à ceux qui vont lancer les travaux de développement de leur projet web. Il vous permettra d'avoir un éclairage objectif des différents aspects de ces frameworks afin que vous fassiez le choix le plus approprié à votre projet et votre équipe.

Toute l'équipe de TheCodingMachine se tient également à votre disposition pour discuter du sujet, concevoir et implémenter vos nouveaux projets !

D'abord, pourquoi utiliser un framework ?

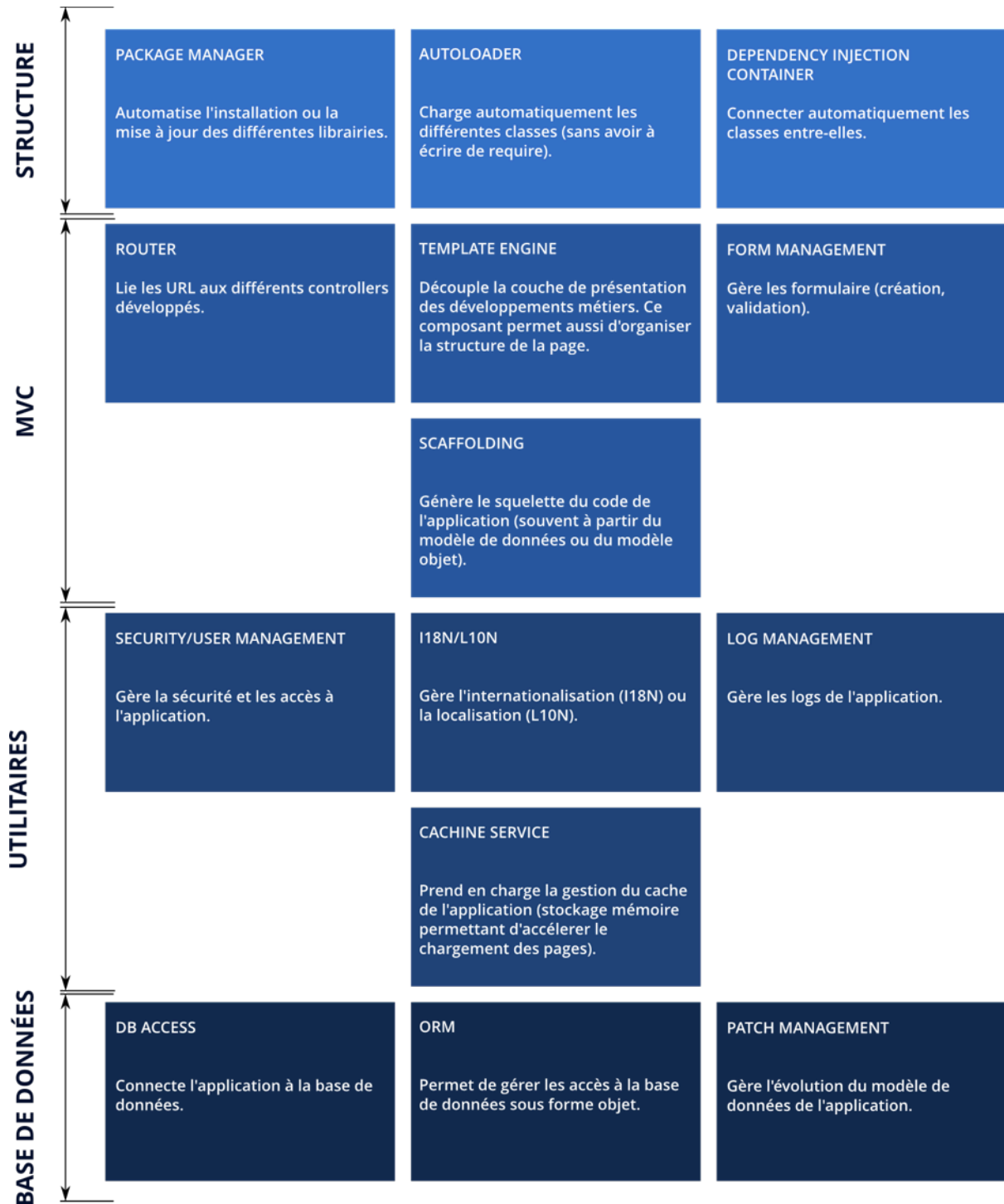
Définition et intérêt d'un framework de développement

Un framework fournit aux développeurs un cadre de travail (un framework quoi ;-)) qui guide l'architecture de développement en conduisant les développeurs à respecter certaines normes. Les frameworks sont conçus pour augmenter la productivité des développements et assurer le respect des bonnes pratiques facilitant ainsi la maintenance et les évolutions d'une application.

Le choix d'un framework s'impose dans la majorité des projets PHP car, au-delà du gain en productivité, il permet d'industrialiser les travaux de développement et de garantir la qualité du code.

Principaux composants d'un framework

La grande majorité des frameworks reprennent les éléments suivants :



Quels sont les frameworks que nous avons testés ?

PHP étant un langage accessible et très apprécié des développeurs web, il existe une multitude de framework.

(cf : http://fr.wikipedia.org/wiki/Liste_de_frameworks_PHP).

Quelques-uns se détachent par leurs fortes utilisations (ce sont tous des frameworks MVC) :

- Symfony 2 : Un des grands framework, il est assez complexe mais intéressant pour les grands projets. Il convient de préciser qu'il est supporté par une équipe française et par conséquent très utilisé en France ;
- Zend Framework 2 : Les créateurs de Zend font partie des développeurs pionniers dans la conception de PHP (rien que ça !). Leur framework, comparable à Symfony 2, est une référence mondiale ;
- Silex : Les concepteurs de Symfony ont créé un framework minimaliste adapté pour les projets de taille raisonnable ;
- Code Igniter : Framework ultra léger (quelques heures suffisent pour apprendre à l'utiliser). Il souffre des défauts de ses qualités, c'est-à-dire qu'il peut être un peu trop simple sur certains projets.

Il en existe bien évidemment pleins d'autres très utilisés. Citons Laravel, Yii, CakePHP... Nous n'avons hélas pas eu le temps de tous les analyser, et cela ne signifie pas qu'ils sont sans intérêt.

Le cas d'école

Nous avons comparé les fonctionnalités des quatre frameworks sélectionnés afin de vous fournir une analyse fine au travers d'un cas concret. Pour ce faire, nous avons effectué un test en situation. Ce test a consisté à chronométrer le temps nécessaire pour réaliser une application de carnet d'adresse, avec les fonctionnalités suivantes :

- Liste des adresses paginée, ordonnable par colonnes ;
- Formulaire de création / modification d'adresse.

Un mot à propos du testeur

Le test a été effectué par David Négrier, notre CTO dans le cadre d'une mission d'audit pour l'un de nos clients. David est le *lead developer* de Mouf, notre framework d'injection de dépendance.

Etant lui-même auteur d'un framework, il n'a aucune raison de favoriser un framework plutôt qu'un autre... à part Mouf bien sûr mais nous ne le testerons pas, nous ne serions pas objectifs.

David dispose d'une forte maîtrise du PHP et de tous les designs patterns associés aux frameworks ainsi que de toutes les bonnes pratiques de développement.

À travers ce test, il a essayé, en toute transparence, de respecter toutes les bonnes pratiques de développement, notamment concernant l'injection de dépendance.

Symfony 2

Présentation

L'un des 2 mastodontes du monde PHP (l'autre étant Zend Framework 2).

Symfony 2 est extrêmement complet. Il s'appuie pour certaines briques sur des librairies externes de qualité (par exemple, l'ORM est géré par Doctrine 2, qui est un projet externe aussi utilisé en option par Zend Framework).

Les composants constituant Symfony 2 sont faiblement couplés et peuvent pour la plupart être utilisés de manière unitaire (voir Silex à ce sujet). Ils sont tous d'excellente qualité et bien documentés (avec une traduction française disponible).

Le « liant » entre les composants est effectué par la notion de « Bundle » et par injection de dépendance. Un bundle est un ensemble de fonctionnalités verticales qui peut être réutilisé. Un bundle peut donc contenir un ou plusieurs contrôleurs, des vues, des entités (objets stockés en base de données), des services, etc.

Cette notion de bundle est légèrement redondante avec la notion de package de Composer (Symfony2 a été développé avant Composer), ce qui peut rajouter de la complexité. De même, la notion de bundle posera souvent des questions que le développeur ne se poserait pas autrement : combien de bundle ? Quelle fonctionnalité mettre dans quel bundle ?

Le gros avantage des bundles est de pouvoir être redistribués. Le site <http://knpbundles.com> référence ainsi 1900 bundles, allant du template Bootstrap au mécanisme de gestion d'utilisateurs standardisé.

Développement de l'application test

0 mn

L'installation se fait par Composer.

Une fois l'installation effectuée, une page effectue une vérification de ma configuration de PHP. Elle m'informe que ma version de APC n'est pas suffisante. Je mets donc à jour APC.

- 9 mn** Je crée un premier bundle en utilisant l'outil en ligne de commande qui m'aide à le mettre en place, puis un contrôleur en suivant le tutoriel. L'utilisation des annotations est élégante et agréable à utiliser. Le référencement du contrôleur dans la configuration n'est pas trivial mais bien documenté (22 minutes)
- 31 mn** Je cherche à disposer d'un template Bootstrap. Je recherche un bundle, en choisit un à jour (d'autres ne fonctionnent qu'avec la version beta) et l'installe. Le bundle requiert « Assetic », un autre bundle extrêmement puissant qui se charge de la gestion des fichiers JS et CSS (et de la compilation des fichiers LESS en CSS). Puissant mais complexe à utiliser. Je finis par réussir à compiler les fichiers LESS et obtenir mon template en 47 minutes.
- 1 h 18** Je passe à la partie base de données. J'utilise Doctrine 2 pour créer un objet qui va générer la table. Cette approche peut s'avérer problématique lorsque le modèle de données est fourni et qu'il faut s'y adapter. L'utilisation de Doctrine est néanmoins très facile. Il me faut moins de 15 minutes pour mettre en place la table, et un mécanisme de sauvegarde. Je développe également un formulaire d'ajout / édition en utilisant Symfony Forms. Le mécanisme est élégant et s'intègre bien avec Doctrine, même s'il posera très certainement de fortes limitations pour les formulaires avancés.
- 1 h 33** Je télécharge un bundle permettant d'afficher les listes avec une pagination. Il existe de nombreux bundle permettant de le faire. L'installation de bundle est toujours un peu complexe, et pas toujours bien documentée suivant le bundle, mais il est facile de trouver des réponses sur internet. Le composant de pagination s'intègre avec Doctrine. 42 minutes
- 2 h 15** A ce stade, j'ai un carnet d'adresse fonctionnel. Cependant, je suis assez surpris par les différents tutoriels. Ils m'encouragent à accéder aux différents composants depuis le contrôleur par leur nom (ServiceLocator pattern). Par exemple, dans le contrôleur, on m'encourage à accéder à un service avec cette ligne de code pour accéder à Doctrine :
- ```
$em = $this->get('doctrine.orm.entity_manager');
```
- Cette approche est un « anti-pattern ». Voir à ce sujet cet article : <http://blog.astrumfutura.com/tag/service-locator/>



Au contraire, le framework devrait être capable d'injecter toutes les dépendances dans le contrôleur avant que je ne l'utilise. Je devrais directement pouvoir utiliser « `$this->em` » pour accéder à l'objet doctrine sans me soucier de donner son nom.

Je me rends alors compte que je dois faire une manipulation pour transformer mon contrôleur en service qui peut être injecté. Cette manipulation est mal expliquée dans la documentation et assez contre-intuitive (alors qu'elle devrait faire partie des étapes logiques de création d'un contrôleur). Je finis par réussir à tout configurer. Temps passé : 53 minutes.

3h08

Je dispose d'une application complète

*Ressenti : Le cœur de Symfony est bien documenté. Utiliser Doctrine est très rapide et vraiment efficace. L'installation de bundles tiers (Bootstrap qui nécessite Assetic) montre la puissance, mais aussi la complexité de Symfony. L'installation d'un bundle est un peu fastidieuse. Je suis surpris de la configuration de l'injection de dépendance (anti-pattern Service Locator utilisé à la place) qui n'est pas encouragée, et qui est vraiment fastidieuse (fichier XML ou Yaml contre-intuitif).*

# Zend Framework 2

## Présentation

Sorti 1 an après Symfony 2, Zend Framework 2 est très semblable à ce dernier. Il est aussi séparé en briques qui peuvent être utilisées unitairement.

Le « liant » entre les composants est effectué par la notion de « Module » (le pendant exact du bundle Symfony 2) et par injection de dépendance. Un module est un ensemble de fonctionnalités verticales qui peut être réutilisé. Un module peut donc contenir un ou plusieurs contrôleurs, des vues, des entités (objets stockés en base de données), des services, etc...

Tout comme pour Symfony, cette notion de module est légèrement redondante avec la notion de package de Composer, ce qui peut rajouter de la complexité. De même, la notion de module posera souvent des questions que le développeur ne se poserait pas autrement : Combien de modules ? Quelle fonctionnalité mettre dans quel module ?

Le site <http://modules.zendframework.com/> référence 470 modules existants.

*Note : contrairement au site dédié aux modules Symfony, il ne met pas en avant les modules phare de ZF2.*

## Développement de l'application test

0 mn

L'installation se fait par Composer.

Le premier contact avec le tutoriel / la documentation de Zend Framework est rude. Le tutorial commence par la mise en place de tests unitaires (avant même une première page). S'il s'agit d'une bonne pratique, elle est dure à suivre et le tutorial l'explique mal. De plus, ce n'est pas particulièrement engageant pour un apprentissage : <http://framework.zend.com/manual/2.0/en/user-guide/unit-testing.html>

Je saute cette étape.

La documentation explique comment créer un module à la main (là où Symfony propose un outil en ligne de commande pour le faire). Pour cela, elle propose de créer 13 (!) répertoires, ainsi que de copier-coller 3 fichiers de configuration cryptiques. Elle ne fait pas référence à l'outil Zend\_Tools qui permet de créer son module en ligne de commande facilement. Je trouve et installe Zend\_Tools et crée mon module sans suivre la documentation (le processus est bien trop complexe à la main). J'utilise Zend\_Tools. Temps écoulé : 9 minutes

**9 mn** Je débute la mise en place du contrôleur. Le framework Bootstrap est inclus dans le package de base de Zend Framework, donc contrairement à Symfony, je n'ai pas à l'installer. La création du contrôleur est rapide. Le mécanisme de configuration des routes est par contre bien moins intuitif que celui de Symfony (pas d'annotations, configuration à la main dans un fichier difficile à comprendre). Temps : 19 minutes

**28 mn** Je passe à la mise en place de la base de données. Bien que Zend Framework 2 puisse s'intégrer avec Doctrine, le tutorial conseille d'utiliser le mécanisme natif ZF2 : le TableGateway. Son utilisation est catastrophique. Nécessite un énorme nombre de ligne de codes /classes pour arriver à faire ce que Doctrine fait en quelques lignes. A éviter absolument. Temps : 24 minutes

**52 mn** Je cherche à afficher une première vue. Le tutorial propose d'utiliser des fichiers PHP pour la gestion du template. Cela place la responsabilité de la sécurité sur le développeur (il ne doit pas oublier de protéger toutes les données entrées par l'utilisateur pour se prémunir des attaques XSS). L'approche Symfony (qui utilise Twig) me paraît meilleure. Réussir à faire fonctionner une vue est complexe (cela nécessite d'éditer un bon nombre de fichiers de configuration). Je fini par mettre en place la liste des adresses, sans pagination. Temps écoulé : 27 minutes

**1 h 19** Je mets en place le formulaire de création/modification d'adresse. L'étape est simple mais assez verbeuse. Je n'arrive pas à mettre en place un widget pour la sélection de la date. Note : l'absence d'annotations comme dans Symfony se fait cruellement ressentir. Les contrôleurs sont moins propres, on ne voit pas les paramètres passés.

Temps écoulé : 5 minutes

**1 h 24** Je tente de mettre en place la pagination. Je trouve plusieurs tutoriaux, mais ils ne sont pas à jour (basés sur une version beta du Zend Framework). Ils ne fonctionnent pas. Je tente de faire le mécanisme de pagination à la main. Je n'y arrive pas, la route ne prend pas en compte mes paramètres, j'ai une erreur et rien dans les logs. Impossible de trouver de l'aide à ce sujet. Temps écoulé (sans résultat probant) : 39 minutes

**2 h 03** Tout comme pour Symfony, je suis surpris. Le tutorial m'encourage à accéder aux différents composants depuis le contrôleur par leur nom (ServiceLocator pattern). Je souhaite que les services soient injectés directement dans le contrôleur. Ceci est possible. Après plusieurs recherches sur internet qui m'amènent sur de mauvaises pistes, je trouve la solution sur le blog de Marco Pivetta : <http://ocramius.github.io/blog/zend-framework-2-controllers-and-dependency-injection-with-zend-di/> Le gestionnaire d'injection de dépendance de ZF2 semble être plus facile à utiliser que celui de Symfony2. On peut injecter les dépendances en utilisant des closures, ce qui est beaucoup plus facile que la configuration XML. Le système ressemble à Pimple (voir Silex plus bas). Temps écoulé : 54 minutes

**2 h 57**

Je dispose d'une application incomplète sans widget pour l'affichage des dates et sans pagination.

*Ressenti : Zend Framework 2 et Symfony 2 se ressemblent beaucoup. Cependant, là où Symfony fait beaucoup pour l'utilisateur (configuration par les annotations, etc.), Zend Framework est plus dur d'accès. Surtout, la documentation est vraiment moins aisée à suivre que celle de Symfony. Le framework en devient cryptique par moment et très peu engageant. Toutes les modifications sont complexes à gérer et on ressent réellement la lourdeur du framework et un manque d'agilité.*

*Alors que Zend Framework 2 propose des fonctionnalités similaires à Symfony 2 (ligne de commande Zend Tools, ORM Doctrine2, etc...), le tutorial n'accompagne pas le développeur dans l'utilisation de ces outils.*

## Présentation

Silex est un « micro-framework ». Basé sur les composants de Symfony 2, il permet de réutiliser facilement les templates Twig, la génération de formulaire de Symfony etc., mais sans la complexité liée aux gros frameworks (pas de bundle ou de modules).

Il dispose d'un mécanisme simple (voir simpliste) d'injection de dépendance : Pimple.

Il n'y a pas de nombreux fichiers de configuration. Toute la configuration de l'application (injection de dépendance et routes) se situe dans le fichier index.php. L'approche de Silex ressemble à Sinatra (Ruby) ou à Express.js (NodeJS) dans la syntaxe de définition des routes.

Ne disposant pas de mécanisme d'ORM propre, j'ai rajouté Doctrine dans le projet Pimple pour gérer les accès à la base.

## Développement de l'application test

|               |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>0 mn</b>   | L'installation se fait par Composer.<br>Le template doit être créé à la main. Je télécharge Bootstrap, jQuery, je crée le template Bootstrap.<br><br>Je configure mon premier contrôleur. Tout comme Symfony et Zend Framework, Silex n'encourage pas la création d'un contrôleur utilisant l'injection de dépendance. La configuration de celui-ci pour supporter l'injection de dépendance est par contre triviale (contrairement à Symfony et ZF2), grâce à Pimple, un mécanisme simple et bien pensé. |
| <b>48 mn</b>  | Mon template est en place, je débute l'installation de Doctrine. L'installation manuelle est plus complexe que l'installation via Symfony 2 (2 fichiers de configuration à créer). Rien d'insurmontable cependant. Temps écoulé : 36 minutes                                                                                                                                                                                                                                                              |
| <b>1 h 24</b> | Contrairement aux 2 autres tests sous Symfony et ZF, je décide d'utiliser une grille Ajax. La grille que je choisis est mal documentée                                                                                                                                                                                                                                                                                                                                                                    |

ce qui me fera perdre du temps. J'utilise le mécanisme de génération de formulaires de Symfony 2 (ce qui me permet de réutiliser quasiment ligne pour ligne le code développé dans le projet Symfony 2). Temps écoulé : 99 minutes.

**3 h 03** Je fais une passe de « propreté », rajout de liens dans la grille... Temps écoulé : 11 minutes

## 3 h 14

## Je dispose d'une application propre

*Ressenti : Silex est réellement agréable à utiliser. On n'est pas perdu dans les différents fichiers de configuration, et on arrive très vite à une solution très propre. Les contrôleurs sont des classes PHP simples sans aucune dépendance sur le framework (découplage maximal).*

# Code Igniter

Les points mis en avant pour ce framework par les développeurs sont :

- Un framework de petite taille ;
- De très bonnes performances ;
- Simple.

Si ce framework connaît un certain succès, il dispose hélas de défauts rédhibitoires :

- Repose sur une **convention de nommage incompatible avec la norme PSR-0** ;
- **Pas de paquet Composer** disponible ;
- **Pas de moteur d'injection** de dépendance ;
- **Pas de moteur de template** par défaut (on peut en rajouter un cependant).

Le carnet d'adresse n'a pas été réalisé pour ce framework, qui est très loin des standards du moment.

## Un petit bonus avec Mouf

Nous n'avons pas fait le test concernant le framework Mouf car nous avons considéré que nous ne pouvions pas être juge et partie, normal on ne renie pas ses enfants ! Sachez simplement que Mouf aurait remporté cet exercice haut la main. C'est le meilleur et le plus beau !

Néanmoins, je ne peux pas m'empêcher de vous ajouter une petite présentation à son sujet.

### Présentation

Mouf est un framework PHP d'injection de dépendances avec une interface de développement web ergonomique pour les développeurs. L'intérêt de Mouf est de permettre aux développeurs de réutiliser des composants qu'ils ont développés ou qui ont été développés par d'autres. Mouf agrège des bibliothèques de composants et vous aide à les lier ensemble. Mouf est en même temps un framework d'injection de dépendance pour la partie graphique et un framework de développement Web complet avec des centaines de paquets disponibles.

De plus, Mouf peut être utilisé en complément d'autres frameworks et s'intégrer dans des environnements tels que Zend, Symfony 2, Drupal, WordPress, etc.

N'hésitez pas à visiter le site web : <http://mouf-php.com>



# Conclusion

L'utilisation d'un framework est aujourd'hui une norme pour les projets professionnels, il vous garantit d'avoir un code maintenable et évolutif. Le choix du bon framework doit répondre aux critères de votre projet (taille du projet, compétence de l'équipe, fonctionnalités souhaitées etc.).

Pour les très grosses applications, Symfony 2 semble prendre l'avantage. Une bonne documentation associée à la présence de nombreux bundles sont les principaux atouts de ce framework. Certains bundles phares, comme Sonata qui permet de gérer les interfaces d'administration facilement, peuvent accélérer considérablement le développement.

Cependant, la tendance actuelle est clairement à la séparation en composants plus petits et plus maniables. Ainsi, Silex, présenté comme un micro-framework pour les petites applications, peut en fait être utilisé pour bien plus, à condition de s'en servir comme routeur, avec les bonnes librairies en support (Doctrine et Twig principalement).

Ainsi, il n'est plus rare de développer des applications en utilisant des composants unitaires d'un framework ou d'un autre. Afin de faciliter l'interopérabilité des différents composants des frameworks entre eux, un groupe de travail - le PHP Framework Interoperability Group (ou PHP-FIG) - s'est formé pour favoriser la création d'interfaces communes. Dans le futur, il est donc probable qu'on opposera moins des frameworks monolithiques, mais qu'au contraire, on les fera travailler ensemble, chacun sur leur point fort.

## **A propos de TheCodingMachine**

TheCodingMachine accompagne ses clients sur des missions de conseil technologique et sur des projets de développement d'applications Web. Nous sommes spécialisés dans le développement de sites Internet, d'intranets, d'applications Web métiers en PHP et en JavaScript.

Fondée, en 2005, par trois anciens consultants technology d'Accenture, TheCodingMachine a piloté plus de 200 projets. Nous travaillons aussi bien pour des grands comptes privés et publics, pour des PME-PMI que pour des startups. Nous avons investi dès notre création dans la R&D, ce qui nous permet par exemple d'être à la pointe des technologies temps réel (NodeJS, BackboneJS, AngularJS, streaming video, chat etc.).

Besoin d'assistance pour votre projet web ? D'un expert technique ? N'hésitez pas à nous solliciter !

[contact@thecodingmachine.com](mailto:contact@thecodingmachine.com)

01 71 18 39 73