

# Introduction à l'industrialisation de PHP

PHP est un langage de programmation largement répandu, probablement le plus utilisé en programmation web. Ce que bon nombre de développeurs PHP ne savent pas, c'est qu'il dispose d'une panoplie d'outils permettant une utilisation confortable de celui-ci en situation professionnelle. Cet article présente un cycle incrémental d'industrialisation possible.

## Cet article explique :

- Le terme industrialisation et ses enjeux.
- Une forme d'industrialisation possible.

## Ce qu'il faut savoir :

- Des connaissances en matière de bonnes pratiques PHP (tests notamment).

L'industrialisation de PHP c'est le processus de fabrication d'une application web en utilisant des outils et techniques qui favorisent la productivité et facilitent l'exécution de certaines tâches critiques. Je m'attarderai sur la phase de développement et plus précisément sur les tests, une partie très importante et souvent la première à outiller lorsque l'on souhaite industrialiser ses développements. Une première partie sera consacrée aux différentes manières de tester une application web en rappelant brièvement pourquoi il est important d'écrire des tests. Suite à cela, quelques outils à connaître (et à mettre en place !) seront présentés. Pour terminer, j'évoquerai d'autres outils présents en amont et en aval de la phase de développement.

## Vous avez dit tests ?

En effet. Tester son application est de loin la meilleure chose à faire. C'est ce qui fera gagner le plus de temps et beaucoup de développeurs croient (encore) le contraire. Le calcul est simple : le temps passé à écrire quelques tests sera bien moindre face au temps passé en débogage.

L'exemple le plus marquant est peut-être celui de la régression de code. PHP excelle en la matière : un petit point-virgule oublié, une portion de code mort, une méthode qui n'est plus appelée, ... Ces erreurs sont fréquentes et quelques tests unitaires permettront de surveiller votre code qui deviendra *robuste*.

Avec le framework Symfony, vous générez vos classes *Model* à chaque fois que vous faites une modifica-

tion dans votre *schema.yml* (le schéma qui représente la base de données). Cependant, ces classes *Model* constituent le *métier* de votre application. Même rigoureux, vous ne pourrez pas toujours répercuter l'ensemble des changements amenés par une modification dans le schéma. L'écriture de tests unitaires permet de surveiller vos classes *Model*, c'est-à-dire le fonctionnement interne de votre application : les accesseurs (getters/setters), les méthodes de calcul et toutes autres méthodes publiques de vos objets. Le débat intéressant est le suivant : doit-on tester les méthodes privées ou protégées ? Oui mais attention, il ne s'agit pas de casser la visibilité de nos méthodes. Une méthode privée ou protégée sera utilisée dans au moins une méthode publique, c'est celle-ci que nous devons tester en faisant en sorte de tester les appels de méthodes privées ou protégées dont elle se sert.

Je viens de vous présenter les tests unitaires, des tests simples garant d'un code robuste, voyons maintenant d'autres types de tests.

Les tests fonctionnels suivent généralement les tests unitaires. Généralement, on distingue deux types de tests :

- Les tests fonctionnels du client qui va voir comment le système réagit ;
- Les tests fonctionnels appliqués au code (les formulaires par exemple) que l'on confond parfois avec les tests d'intégration.

```
~/work/jobeeet $ php symfony test:unit Jobeeet
1..8
# ::slugify()
ok 1 - ::slugify() converts all characters to lower case
ok 2 - ::slugify() replaces a white space by a -
ok 3 - ::slugify() replaces several white spaces by a single -
ok 4 - ::slugify() replaces non-ASCII characters by a -
ok 5 - ::slugify() removes - at the beginning of a string
ok 6 - ::slugify() removes - at the end of a string
ok 7 - ::slugify() replaces the empty string by n-a
not ok 8 - ::slugify() replaces a string that only contains non-ASCII ch
# Failed test (/Users/fabien/work/symfony/dev/1.2/lib/vendor/limelime/
# got: ''
# expected: 'n-a'
Looks like you failed 1 tests of 8.
~/work/jobeeet $
```

Figure 1. Exemple de tests unitaires avec Lime (Extrait de Jobeeet – [www.symfony-project.org](http://www.symfony-project.org)).

```
~/work/jobeeet $ ./symfony test:functional frontend categoryActions
# get /category/index
ok 1 - request parameter module is category
not ok 2 - request parameter action is index
# Failed test (/Users/fabien/work/symfony/dev/1.2/lib/test/sfTesterRequest.class.php at line 48)
# got: 'show'
# expected: 'index'
not ok 3 - status code is 200
# Failed test (/Users/fabien/work/symfony/dev/1.2/lib/test/sfTesterResponse.class.php at line 257)
# got: 404
# expected: 200
ok 4 - response selector body does not match regex /This is a temporary page/
1..4
Looks like you failed 2 tests of 4.
~/work/jobeeet $
```

Figure 2. Exemple de tests fonctionnels avec Symfony (Extrait de Jobeeet – [www.symfony-project.org](http://www.symfony-project.org)).

Les tests d'intégration forment une troisième famille de tests. Ils visent à tester la cohésion de l'application lorsqu'on y ajoute de nouvelles fonctionnalités, modules, etc.

### Des outils pour les tests !

Pour tester une application, il faut avoir les outils qui vont bien et en PHP, il y en a rassurez-vous. Voici deux bibliothèques incontournables pour nos tests unitaires :

- **Lime** : framework de tests embarqué dans Symfony 1.x ;
- **PHPUnit** : semblable à **JUnit** en Java, il tend à devenir le framework principal de tests pour PHP. Je ne peux que vous le recommander. Ce dernier vous permettra de profiter de sur-couches issues du monde Java pour vos développements PHP comme nous allons le voir par la suite.

Attention, la rédaction des tests unitaires ne sera pas traitée dans cet article. Passons donc aux tests fonctionnels.

Symfony embarque un mini framework de tests fonctionnels. Celui-ci permet de faire des tests qualifiés de *techniques*. En d'autres termes, nous allons tester des formulaires, des navigations de pages et autres mais sans pour autant profiter d'un vrai rendu et de toutes les interactions possibles avec un vrai navigateur. En effet, la classe *sfBrowser* de Symfony ne fait que des requêtes HTTP sans possibilité de tester les requêtes AJAX par exemple.

Pour cela, il existe un outil très pratique nommé : **Selenium**. C'est une application disposant de plusieurs éléments dont un plugin pour Firefox permettant d'enregistrer un scénario de tests et de le rejouer par la suite. C'est puissant puisque, une fois enregistré, nous pouvons tester à peu près tout ce que l'on souhaite et le rejouer sur plusieurs navigateurs sans souci. Vous voyez aisément l'intérêt ?

```
~/work/tmp $ ./symfony test:coverage --detailed test/unit/model/ArticlePeerTest.php lib/model/ArticlePeer.php
>> coverage running /Users/fabien/work/tmp/...model/ArticlePeerTest.php (1/1)
lib/model/ArticlePeer                                     75%
# missing: 16
TOTAL COVERAGE: 75%
~/work/tmp $
```

Figure 3. Exemple de couverture de tests avec Lime (www.symfony-project.org).

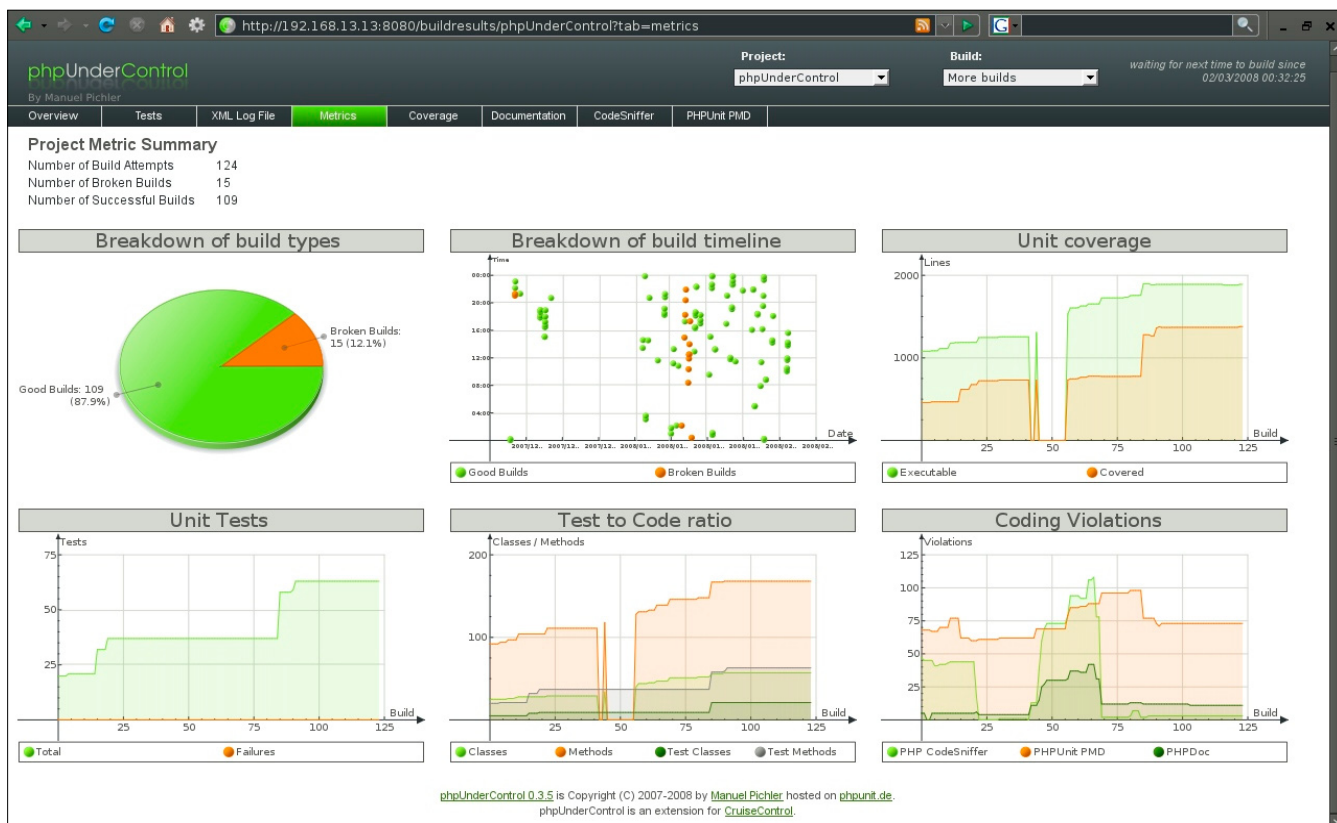


Figure 4. Exemple de graphiques dans phpUnderControl.

En complément, nous avons le test de couverture de code. Ce test quantifie le code testé avec les tests unitaires, autrement dit l'efficacité de ces tests. Cette couverture se mesure avec **Xdebug** couplé à l'un des deux framework de tests énoncés ci-avant.

Il nous est possible de faire des tests unitaires et fonctionnels mais leurs exécutions restent manuelles. Il serait bon d'affranchir nos développeurs du fait de lancer nos tests. Il existe des outils pour cela, regroupés autour du terme *intégration continue*.

## L'intégration continue

*Continuous Integration* (CI) est l'action d'automatiser les tests et de les lancer périodiquement. Le logiciel employé est appelé serveur d'intégration continue, en voici deux :

- **phpUnderControl** : qui utilise en sous-couche *CruiseControl*, le serveur CI Java ;
- **Hudson** : directement emprunté à Java.

**phpUnderControl** (Figure 4) est très sexy mais compliqué à installer. Un projet vise à le rendre plus simple

de prise en main notamment par le biais d'un paquet officiel Debian.

**Hudson** (Figure 5) quant à lui est très simple, doté d'énormément de plugins et, bien qu'outil Java, très utilisé dans le monde PHP professionnel.

Un serveur CI se connecte à un gestionnaire de versions et possède la capacité d'effectuer une série de tâches périodiquement. On pourra, par exemple, choisir de lancer nos tests après chaque `commit` ou à intervalle de temps spécifié. Nous reviendrons sur les gestionnaires de versions dans la dernière partie de cet article.

Brancher nos tests unitaires et fonctionnels est très simple puisque **Lime** et surtout **PHPUnit** reprennent le format de sortie de **JUnit**, une référence en matière de tests unitaires Java.

Cependant, le vocabulaire est peu commun, il faudra d'abord se familiariser avec le terme `build` qui signifie étape de création. Autrement dit pour une application PHP, la récupération des sources depuis un gestionnaire de versions, l'exécution des tests, des analyses sur le code, la remontée d'indicateurs ou métriques et l'intégration avec d'autres outils extérieurs que je présenterai plus loin.

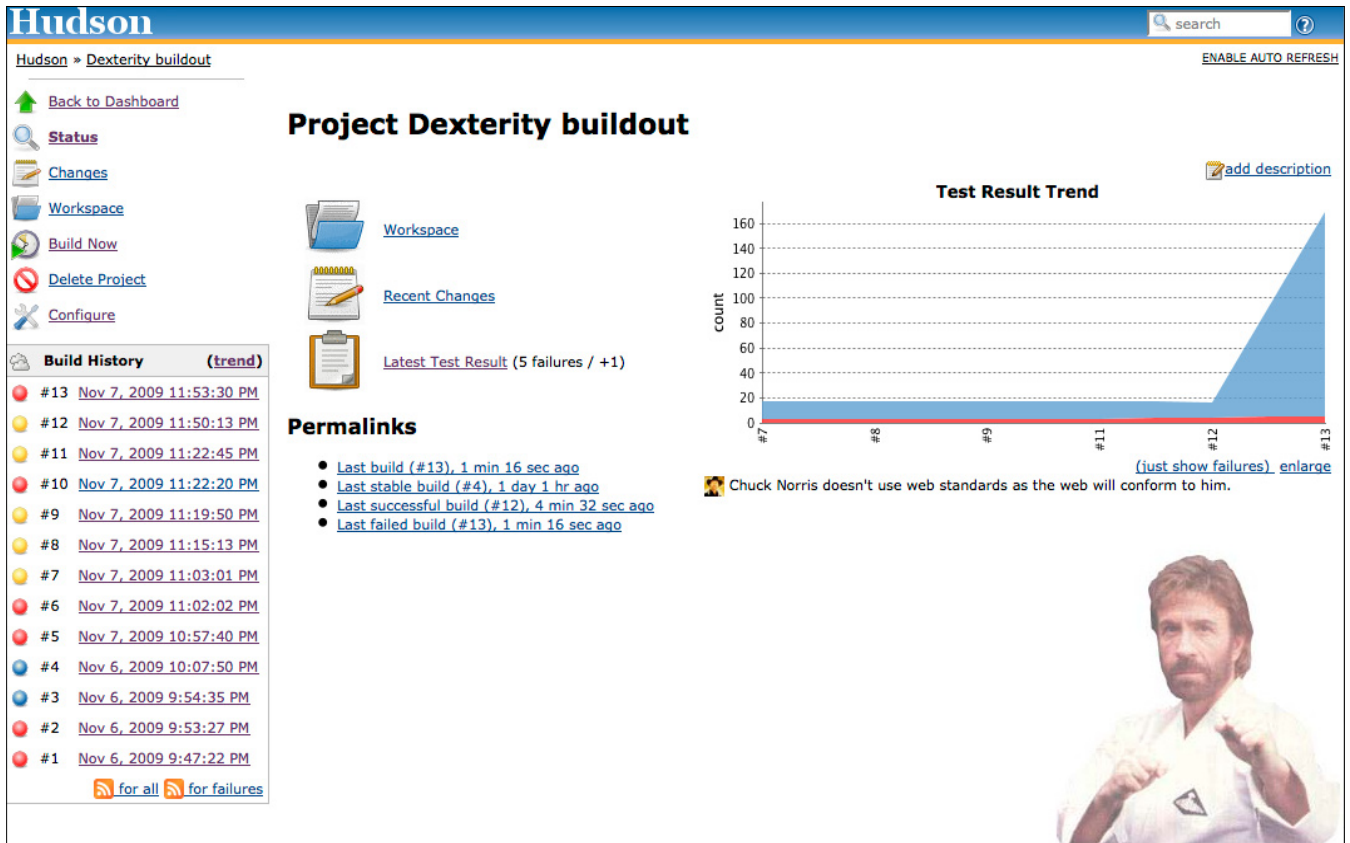


Figure 5. Visualisation des résultats dans Hudson (Capture provenant de [www.martinaspeli.net](http://www.martinaspeli.net)).

Mais que sont ces analyses sur le code ? C'est l'objet de ma prochaine partie.

### Analyser pour mieux régner

Lorsque notre serveur CI est fonctionnel, il est intéressant de lui ajouter des fonctionnalités qui seront lancées après l'exécution de nos tests et la réussite de ceux-ci. Analyser le code source est une bonne idée. Il existe trois niveaux d'analyse :

- L'analyse lexicale : pour le code redondant, les erreurs dans le suivi du guide de style ;
- L'analyse syntaxique : pour les erreurs de paramètres, code mort, non respect de règles Objet ;
- L'analyse du bytecode PHP (opcode).

Au niveau lexical, on dispose de **PHP\_CodeSniffer** qui va vérifier le respect de règles prédéfinies. L'utilisation courante est d'écrire une série de règles définissant son guide de style et de valider notre code source. Si une portion de code ne respecte pas notre guide, **PHP\_CodeSniffer** l'indiquera dans son rapport. Couplé à **Hudson** et à son plugin **Checkstyle**, nous aurons directement l'information. C'est très pratique lorsque plusieurs développeurs travaillent sur un même projet et disposent du guide de style de l'entreprise.

Au niveau syntaxique, **PHP\_Depend** va nous fournir des informations sur le niveau d'abstraction de notre code, le nombre moyen de méthodes par classe, la

complexité cyclomatique, le nombre de classes héritées et bien d'autres. Notez que **PHP\_Depend** est l'équivalent PHP de **JDepend** en Java.

On dispose également de **PHP Mess Detector** qui est proche de **PHP\_Depend** et qui lui va détecter les portions de code mort ou encore le nombre excessif de paramètres pour une méthode. On utilisera le plugin **DRY d'Hudson** pour visualiser les résultats.

Troisième et dernier outil, **PHP Copy/Past Detector** qui va détecter la duplication de code. Un *must have* lorsque l'on suit la logique *Don't Repeat Yourself !*.

Analyser notre code au niveau byte code est plus compliqué, les outils sont moins connus et je ne m'y attarderai pas. Pour les intéressés, voyez **Vulcan Logic Disassembler**, **Bytekit** et **Bytekit-cli**.

L'ensemble des outils présentés dans cette partie ne nous servent à rien si l'on ne peut pas exploiter les indicateurs retournés. Pour visualiser les informations, on peut utiliser directement **Hudson** ou **phpUnderControl** mais il existe également **Sonar** pour PHP (Figure 6). **Sonar** est l'arme ultime en Java pour l'analyse de code et il est porté pour PHP, foncez ! Certaines métriques (celles de **PHP\_Depend** par exemple) ne seront pas très explicites dans **Hudson**, il sera préférable de connecter **PHP\_Depend** à **Sonar** et **Sonar** à **Hudson**.

Dernier outil de cette partie : **phpDocumentor** qui va générer une documentation complète de l'application en se basant sur les commentaires dans les sources. Bien évidemment, si des erreurs sont présentes

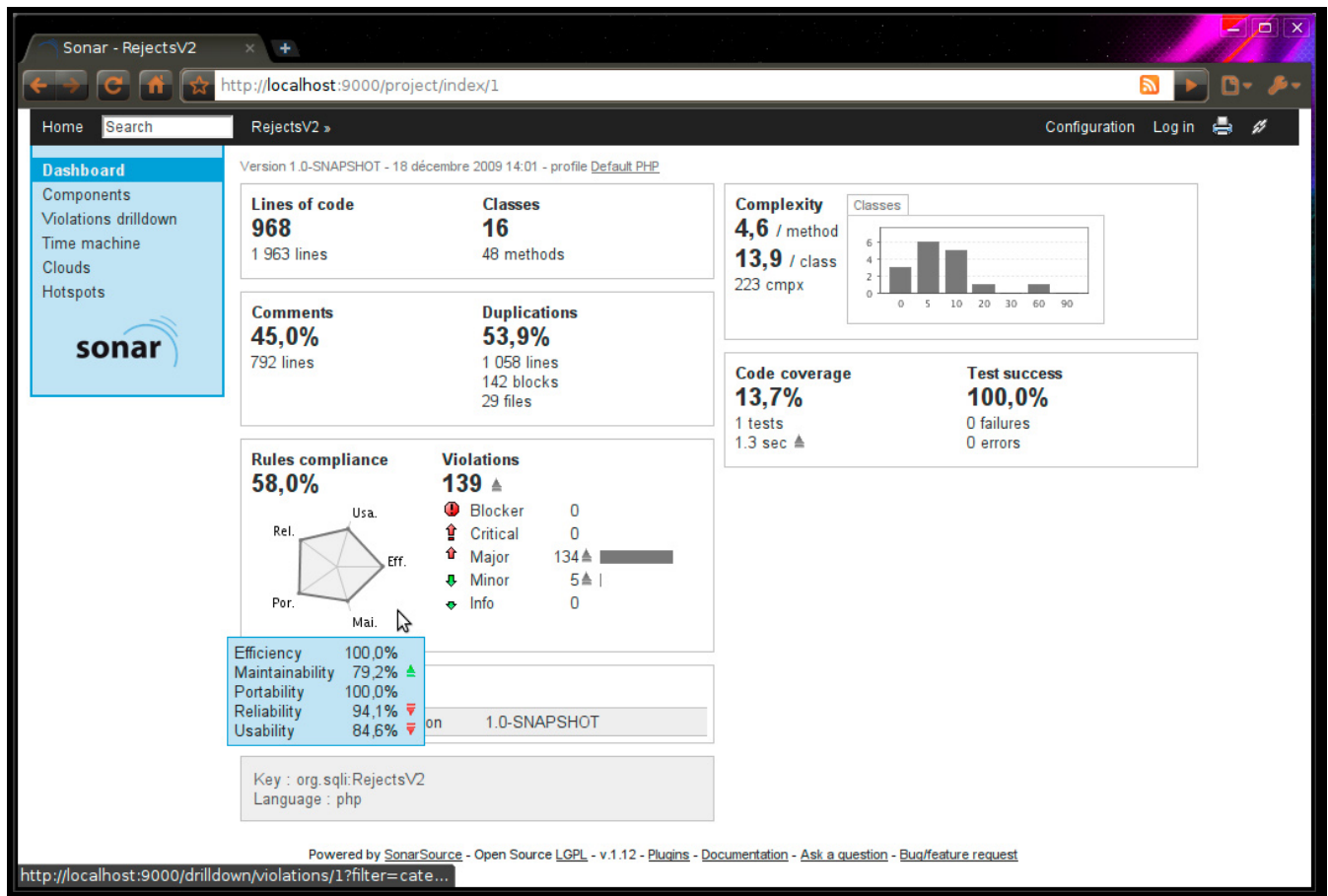


Figure 6. Dashboard de Sonar PHP (Capture provenant de [www.industrialisation-php.com](http://www.industrialisation-php.com)).

## Sur Internet

- <http://symfony-project.org/> : Site officiel du framework Symfony,
- <http://www.phpunit.de/> : Site du projet PHPUnit,
- <http://seleniumhq.org/> : Site officiel de Selenium,
- <http://www.xdebug.org/> : Site officiel de Xdebug,
- <http://phpundercontrol.org/> : Site du projet phpUnderControl,
- <http://hudson-ci.org/> : Site officiel du serveur d'intégration continue,
- [http://pear.php.net/package/PHP\\_CodeSniffer](http://pear.php.net/package/PHP_CodeSniffer) : Page du projet PHP\_CodeSniffer,
- <http://pdepend.org/> : Site officiel de PHP\_Depend,
- <http://phpmd.org/> : Site officiel de PHP Mess Detector,
- <https://github.com/sebastianbergmann/phpcpd> : Github du projet PHP Copy/Past Detector,
- <http://derickrethans.nl/projects.html> : Documentation de Vulcan Disassembler Logic,
- <http://www.bytekite.org/> : Site officiel de Bytekit,
- <https://github.com/sebastianbergmann/bytekite-cli> : Github du projet Bytekit-cli,
- <http://sonar-php.sqli.com/> : Démo de Sonar pour PHP,
- <http://www.phpdoc.org/> : Site officiel de phpDocumentor,
- <http://git-scm.com/> : Site officiel de l'excellent gestionnaire de versions,
- <https://github.com/capistrano/capistrano> : Github du projet Capistrano,
- <http://symfony-project.org/> : Site officiel de l'ORM PHP Doctrine,
- <http://www.mantisbt.org/> : Site officiel du bug-tracker MantisBT,
- <http://www.bugzilla.org/> : Site officiel du Bugzilla,
- <http://www.industrialisation-php.com/> : Blog sur l'industrialisation PHP.

dans les commentaires (exemple : s'il n'y en a pas ...), l'outil fera remonter l'information.

La plupart de ces outils seront mis en place via **Phing**, un automatisateur de tâches open source comme peut l'être Ant en Java. On utilise un fichier XML nommé

*build.xml* contenant une série d'actions possibles pouvant être dépendantes les unes des autres.

Nous avons à présent totalement défini les outils supplémentaires attachés à notre build. Elle est complètement configurée, les résultats des analyses sont

visibles. Nous pouvons désormais travailler en amont et en aval de celle-ci. C'est ce qui suit.

### Préparons et automatisons

Nous sommes dans une dynamique d'industrialisation, c'est-à-dire simplifier et automatiser les processus historiquement lourds et difficiles à gérer. Dans le cadre d'une application web, fiabiliser notre code est très important mais ne va pas directement impacter notre processus de conception. La fiabilisation agira plutôt sur les délais de développement et réduira les phases de débogage.

En amont de cette fiabilisation du code, nous pouvons mettre en place un outil très simple, très connu mais pas toujours utilisé : le gestionnaire de versions, permettant de conserver l'historique intégral des modifications apportées à notre code. Les gestionnaires de versions à la mode tels **Git** ont la notion de *branches* permettant d'explorer des pistes sans pour autant perturber l'ensemble majeur des sources de l'application. Tout projet devrait être versionné...

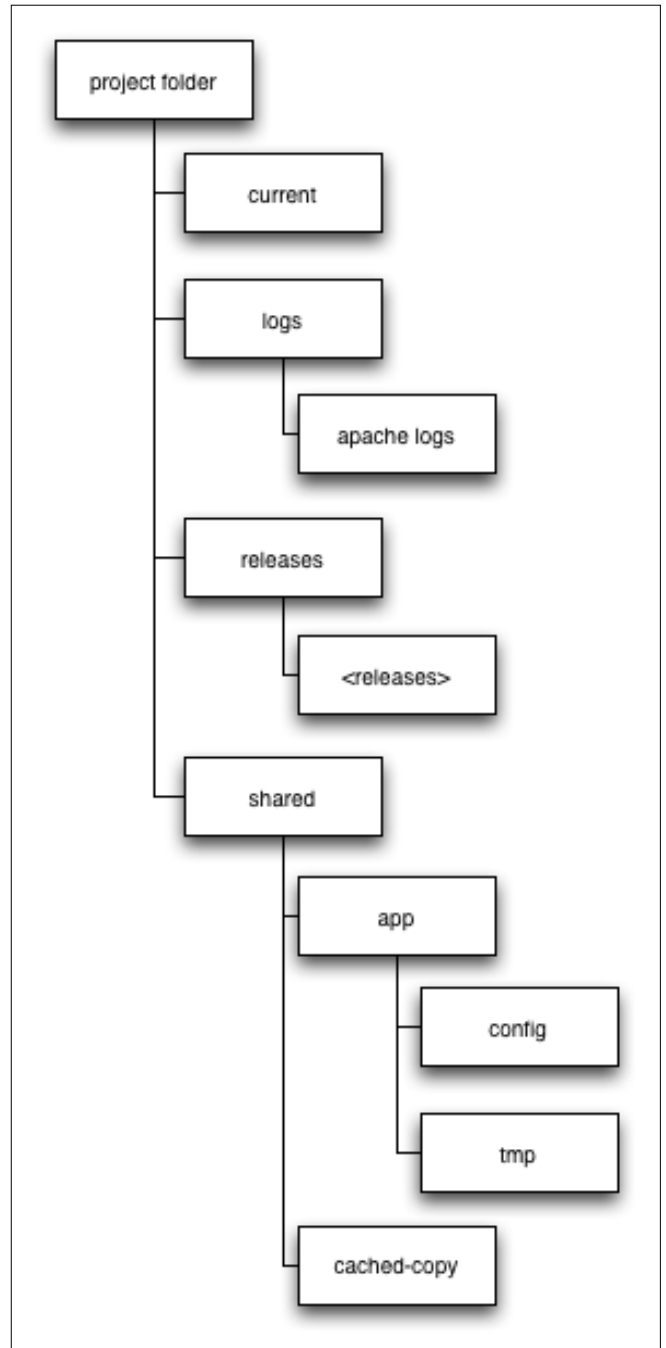
Ensuite, nous pouvons définir des environnements de développement :

- Développement : utilisé par les développeurs avec un maximum d'informations, de remontées d'erreurs, etc ;
- Test : utilisé par les outils de test et d'analyse. Cet environnement peut voir sa configuration varier pour améliorer le processus de tests : utiliser un moteur de base de données différent de celui en production pour tester la fiabilité des requêtes SQL ou de l'ORM ;
- Pré-production : c'est l'environnement de production répliqué à J -1, il peut être utile pour tester le déploiement ;
- Production : c'est l'environnement client, là où nous n'avons plus vraiment le droit à l'erreur.

Après une build réussie, nous obtenons une application fonctionnelle. On peut alors se poser la question du passage en production : ce que l'on appelle *déploiement*. Des outils existent pour faciliter cette phase critique.

**Capistrano** est le premier. C'est un outil écrit en Ruby et initialement fait pour déployer des applications *Ruby On Rails*. Le fonctionnement est simple : on initialise une configuration pour un projet donné, on configure nos accès serveurs, gestionnaire de versions, base de données, ... Éventuellement, nous pouvons surcharger certains comportements. **Capistrano** sait gérer autant de serveurs qu'on le souhaite, il va exécuter une succession de commandes sur chaque serveur : récupération des sources, liaison à la base de données, effacement des caches, rotation des logs, etc.

**Capistrano** met en place une arborescence de fichiers qui lui est propre sur chaque serveur web (Figure 7). Le dossier *releases* contient l'ensemble des versions



**Figure 7.** Arborescence créée par Capistrano sur chaque serveur web distant.

déployées du projet, le dossier *shared* contient les éléments partagés entre toutes les *releases* et un lien symbolique *current* pointe sur la dernière version déployée de l'application. Ce système de releases permet de faire un *roll back* si un problème survient lors du déploiement.

Enfin, **Capistrano** pourrait gérer les migrations des bases de données mais il faut faire attention.

Une migration de base de données est très critique car elle a un impact directement sur l'intégrité des données : une requête plante et toute la base peut être cassée. Pour éviter cela, on peut soit faire les migrations à la main, soit faire confiance à certaines applications PHP. **Doctrine** sait gérer les migrations et le

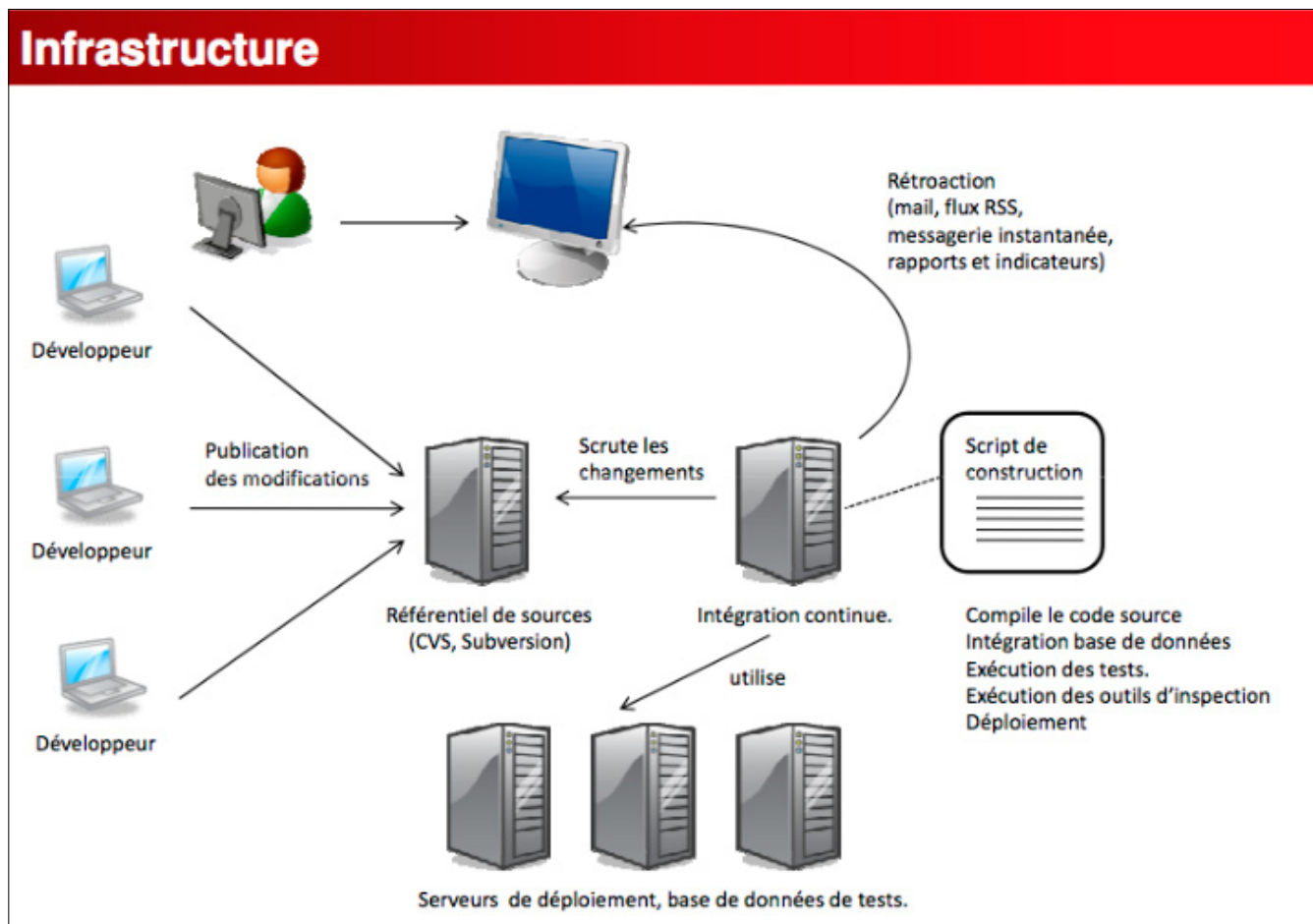


Figure 8. Schéma global de l'architecture en place (Crédit : Jean-Luc Nta)

fait plutôt bien, moyennant la relecture des scripts de migration générés. Là encore, il faut soit être de nature confiant, soit passer les migrations à la main.

Notre application est maintenant déployée, accessible aux utilisateurs finaux. Pourtant, des problèmes surviennent et les utilisateurs nous submergent de mails. Un type d'application est dédié à cette récolte de retours utilisateurs (*feedbacks*): les *bug-trackers* ou encore rapporteurs d'erreurs. Les utilisateurs saisissent directement leurs mésaventures via une interface web et sont tenus au courant par mail des avancées : demande de précisions, résolution, rejet, prise en compte, ...

**MantisBT** est une solution répandue, qui peut se connecter à un serveur **Hudson** et recevoir des informations lorsque les erreurs sont corrigées dans une build. Il existe également **Bugzilla** utilisé par Mozilla, plus complet et donc moins simple à prendre en main.

## Conclusion

Après cette introduction qui présente de nombreux outils, que pouvons-nous en retenir ? Premièrement que nous disposons de beaucoup d'outils et qu'il ne reste qu'à s'en servir. Il existe un bon nombre d'articles techniques sur le sujet.

La progression décrite ici est une logique d'industrialisation : commencer par écrire des tests, les auto-

matiser puis mettre en place des outils d'analyse et les automatiser. Après cela, on va pouvoir changer d'autres phases : un gestionnaire de version pour centraliser les sources, différents environnements de développement, une automatisation du processus de déploiement et une gestion des retours clients.

S'industrialiser est un fait, de nombreuses entreprises s'y attèlent. De plus, les frameworks actuels facilitent les choses, les concepts agiles sont bien en phase avec l'industrialisation et PHP est clairement professionnel.

L'unique problème reste la gestion du changement, qui n'est ni lié à PHP, ni lié au monde informatique mais intimement mêlé au genre humain. Il faudra alors préparer l'équipe aux changements, montrer l'efficacité et la nécessité de s'industrialiser. Une industrialisation réussie est une industrialisation convaincante et performante.

## WILLIAM DURAND

L'auteur est élève ingénieur et développeur web indépendant spécialisé dans le développement PHP avec le framework *Symfony* et *Java* depuis près de deux ans. Il est également fondateur de *Bazinga*, une agence spécialisée en développement web et rythmée par les pratiques agiles.

Contact direct : <http://www.willdurand.fr>