

symfony a SENSIO LABS * event
PARIS2009
11-12 Juin



Les bonnes pratiques du développement symfony en 30 points clés

Marc Hugon, Gilles Taupenas

Enjeux



- L'existant PHP4
- Les outils actuels
 - PHP5
 - Symfony 1.2 avec Propel / Doctrine
- Opportunité pour les applicatifs
 - Développement rapide
 - Pérennité



PHP4



- Fin de vie : « enterré » depuis le 8/08/08
- Code essentiellement procédural
- Support objet pauvre
- Pas de dynamique pour faire du développement professionnel



PHP5 seul



- Bon support objet
 - Héritage
 - Introspection
 - Système d'interface
- Outils pratiques
 - SPL : Autoload, ...
- Les bons projets full PHP5 existent mais ils coûtent cher à maintenir : ils embarquent leur propre formalisme



PHP5 / symfony



- Formalisme très présent
 - Un projet symfony quelconque a une arborescence de base
 - Trop contraignant ?
 - Symfony peut être paramétré pour fixer une logique particulière et nécessaire
 - En général, c'est le code existant qui contraint à tordre symfony
- Outils
 - Ils sont dans symfony car ils constituent des fonctionnalités globalement nécessaires aux applications web
 - Définissent un écosystème :
 - Ils communiquent entre eux
 - Ils sont testés pour bien fonctionner seuls et entre-eux



Développement rapide



- La présentation est là pour montrer que les fonctionnalités permettent
 - De gagner du temps en développement, en intégration et en tests
- Et surtout pas au détriment de la qualité
 - Ne pas créer du code peu ou pas maintenable



MVC



- Modèle – Vue - Contrôleur
- Créé en 1979
- Permet un bon découpage macroscopique entre les 3 grandes entités d'une application
 - Modèle : constitue la logique métier effectuant le traitement des données
 - Vue : gère les interactions avec l'utilisateur
 - Le contrôleur : indispensable mais doit rester léger
 - Contrôle les données utilisateur
 - Coordonne les opérations de la vue et du modèle



Pourquoi MVC?



- Découpage de l'application
- Répartition du travail des intervenants

Adaptable à toute application !

- Importance primordiale
- Cas de non-respect du découpage MVC est l'erreur majeure rencontrée en audit



Avantages du respect de MVC



Pouvoir passer facilement

- d'un client ligne de commandes à un client graphique / web
- D'un client navigateur web à un service web (passage en flux business XML/SOAP)

Adaptation aux capacités du client web

- Si Flash présent : choisir flash
- Sinon : si javascript présent : choisir javascript
- Sinon : choisir version HTML sans javascript



Les bonnes pratiques



MVC



MVC : le respecter



BONNE PRATIQUE 1

- Comment être sûr de respecter le MVC?
 - Se poser la question de l'emplacement des parties du code
 - Contrôle du code régulier
 - Refactoriser

- Garder propre chez soi est une rigueur de tous les instants



MVC : Taille du contrôleur



BONNE PRATIQUE 2

- Que fait le contrôleur ?
 - Met en relation le modèle et la vue
 - Gère les événements de l'utilisateur
- Ce que ne fait pas le contrôleur
 - Il ne connaît pas les règles métiers
 - Il n'est pas intelligent
- Le code PHP d'une méthode du contrôleur (executeXXX), c'est 10 lignes maximum!



MVC : Bien utiliser les composants



BONNE PRATIQUE 3

- Composants (component)
 - Il s'agit de reproduire un schéma MVC propre cloisonné à un besoin réutilisable
 - Un composant n'est pas créé directement, on crée d'abord un partial
 - Si ce partial est réutilisé et qu'il doit contenir du code PHP, alors on peut le transformer en « component »
- Refactoring is a key point (and KISS)
 - Faire un partial si c'est suffisant
 - Le transformer en composant plus tard



MVC : Bien utiliser les helpers



BONNE PRATIQUE 4

- **Helpers**

- **Standard** : mise en forme facile de HTML

```
<?php use_helper('ProjectTags');?>
```

```
<?php echo clickable_image($user->getAvatar(),  
    '@profile_info?'. $user->getId()) ?>
```

- **Classes Helpers** : permettre de factoriser une logique de pivot pour

- Récupérer un format standard provenant du modèle
 - Adapter à ces données au format vue



MVC : PHP dans les templates



BONNE PRATIQUE 5

- Il en faut le moins possible !
- Template = Vue = HTML

Eviter

```
<?php echo '<p>'.$title.'</p>' ?>
```

Préferer

```
<p><?php echo $title ?></p>
```



Les bonnes pratiques



Bases de données



Bases de données : utiliser un ORM



BONNE PRATIQUE 6

- Choisir Propel ou Doctrine
- S'affranchir de la base de donnée cible
 - Travailler sur des bases de données différentes selon l'environnement
 - Permettre de faire évoluer le modèle sans « casse »
- Abstraire en objet les manipulations en base de données
 - Surcharge
 - Permettre de changer la structure et le comportement



ORM : pas d'outil propriétaire



BONNE PRATIQUE 7

- Deux choix existants (Doctrine, Propel)
- Utiliser un ORM propriétaire
 - Réinventer la roue
 - Maintenance à prendre en charge
 - Documentation à créer
 - Transfert de compétence à assurer
- Pour des besoins spécifiques, étendre l'ORM standard choisi



ORM : requêtes SQL



BONNE PRATIQUE 8

- Ne doivent pas exister dans le code
- Si elles existent (mais elles n'existent pas), elles ne sont ni dans la Vue ni dans le Contrôleur
- Si elles existent (mais elles n'existent pas), c'est pour utiliser des méthodes spécifiques à la base de données cible non gérées par l'ORM
 - Procédures stockées
 - Données de géolocalisation dans PostgreSQL



ORM : fonctions comportementales



BONNE PRATIQUE 9

- Existent en Propel et Doctrine
- Permettent de
 - donner un comportement
 - hors système d'héritage (=nature)

Exemples : versioning, liste arborescente, tag

- Mais point trop n'en faut : bénéfice ne doit pas dépasser le coût de la maintenance d'entrelacement des cas limites
- Trop de comportements sur une même classe
⇒ Risque de cas limites



ORM : gestion unifiée du schéma



BONNE PRATIQUE 10

- Référentiel unique
- Niveau 1 : schéma unique
 - YML ou XML
 - YML plus lisible humainement
 - Mais XML facile à tester valider
 - But
 - Un seul fichier important pour construction des classes de base du modèle
 - Maintenir à jour ce schéma et ses données de base : facile à installer



ORM – gestion unifiée du schéma



BONNE PRATIQUE 10

- Référentiel unique toujours !
- Niveau 2 : schéma de la base modélisé
 - Outil : DBDesigner, MySQLWorkbench, ...
 - On versionne ce fichier et non plus le schéma YAML ou XML
 - Il est régénéré pour DBDesigner4 grâce à
 - sfDbDesignerPlugin pour Doctrine
 - sfDb4ToPropelPlugin pour Propel



ORM : maintenances des bases



BONNE PRATIQUE 11

- SQL seul pour faire évoluer une base en production?
- Et si plusieurs instances évoluent différemment?
- Fonctionnalités de migration
 - Comment ?
 - Propel : sfPropelMigrationsLightPlugin
 - Doctrine : de base
 - Intelligence : gestion de metadonnées : numéro de version



Les bonnes pratiques



Maintenance



Maintenance : lisibilité du code (1/5)



BONNE PRATIQUE 12

(Se re-)trouver facilement dans un projet

- Acteurs : développeurs, intégrateurs
- Rendre lisible => homogénéiser
 - Standards de codage
 - PHP : OOP, syntaxe alternative
 - Côté Symfony :
 - Arborescence : elle qualifie ce qu'elle contient
 - Outils :
 - OOP, autoloading : trouver les classes, celles dérivées...
 - Templating
 - Limite : attention au critère de dispersion



Maintenance : lisibilité du code (2/5)



BONNE PRATIQUE 13

- Le code HTML ne peut se trouver que dans des templates
- Aucune méthode ne doit contenir par exemple

```
function forbiddenMethod($value = 0)
{
    $content = '<p>The value is `';
    $content .= ($value > 0)? ` more than 0' : ` exactly 0';
    $content .= '</p>';
    return $content;
}
```

Réactions ?



Maintenance : lisibilité du code (3/5)



BONNE PRATIQUE 13

- Une norme unique doit être utilisée dans le code PHP
 - Règles d'écritures (indentation, syntaxe des variables, langue utilisée)
 - Documentation, entêtes
 - Règles de retour à la ligne

```
function myFunction($var)
{
    if ($test)
    {
        //do something
    }
}
```



Maintenance : lisibilité du code (4/5)



BONNE PRATIQUE 13

- Les templates doivent respecter les standards d'intégration
- Ils peuvent être utilisés par les intégrateurs

Syntaxe classique

```
<?php foreach ($cars as $car) {?>
    <? php echo $car->getModel(); ?>
    <?php if ($car->hasTurbo()) { ?>
        (turbo)
    <?php }} ?>
```

Syntaxe alternative

```
<?php foreach ($cars as $car) ?>
    <? php echo $car->getModel(); ?>
    <?php if ($car->hasTurbo()): ?>
        (turbo)
    <?php endif;?>
<?php endforeach; ?>
```



Maintenance : lisibilité du code (5/5)



BONNE PRATIQUE 13

Penser sémantique mais comment ?

- XHTML seulement ? pour les intégrateurs !
- Symfony ? Seulement pour les développeurs !

Symfony permet d'aller plus loin avec : les partials, les composants

Plus de sémantique => maintenance accrue

Et les intégrateurs arrivent à faire du symfony ?

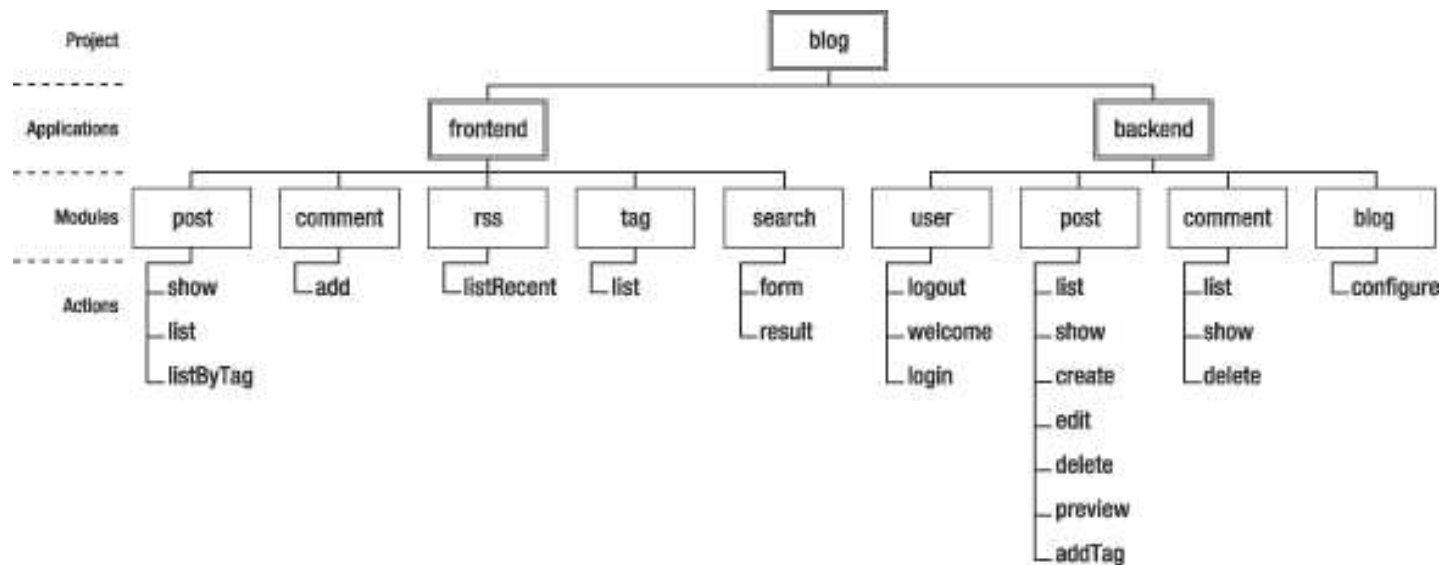


Standards de code : répertoires



BONNE PRATIQUE 14

- Pour les librairies projet, utiliser le répertoire « lib », éviter le répertoire apps/xxx/modules/yyy/lib
- Pour les libraires externes, utiliser le répertoire « lib/vendor »



Maintenance : tests (1/2)



BONNE PRATIQUE 15

- Pendant la phase de production, écrire en priorité des tests unitaires

```
$t = new lime_test(3, new lime_output_color());  
  
$t->is(Jobeet::slugify('Sensio'), 'sensio');  
$t->is(Jobeet::slugify('sensio labs'), 'sensio-labs');  
$t->is(Jobeet::slugify('paris,france'), 'paris-france');
```

- Un test s'écrit le plus tôt possible
- Ne pas viser une couverture complète de l'application



Maintenance : tests (2/2)



BONNE PRATIQUE 15

- A la fin de la phase de développement, écrire des tests fonctionnels

```
$browser->  
  get('/category/index')->  
    with('response')->  
      begin()->  
        isStatusCode(200)->  
          checkElement('body', '!/This is a temporary page/')->  
            end() ;
```

- Maintenir les tests
 - Lancer les tests à chaque « commit »
 - Corriger immédiatement les tests qui échouent



Maintenance : utiliser symfony



BONNE PRATIQUE 16

Il ne faut pas bypasser le framework, ni le refaire

- Pas de `$_SESSION`, utilisez `sfUser`
- Pas de `$_SERVER`, `$_POST`, `$_GET`, utilisez `sfRequest`
- Pas de variables globales



Maintenance : code Symfony

BONNE PRATIQUE 17



Le code du framework ne doit **jamais** être modifié
– Mais il peut être surchargé !

Conseil : systématiquement utiliser symfony en
SVN:externals pour

- Symfony
- Plugins



Maintenance : fixtures

BONNE PRATIQUE 18



- Avoir des jeux de données permet d'installer facilement l'application sur un serveur de développement sans contrainte forte
- Il faut donc avoir des fixtures pour assurer le fonctionnement à partir d'un environnement vierge
 - Créer des jeux de données pour chaque nouvelle fonctionnalité
 - Maintenir les jeux de données de test à chaque évolution
 - Utiliser ces jeux de données dans les tests



Maintenance : utiliser le routing (1/3)



BONNE PRATIQUE 18

- Ne JAMAIS utiliser d'URLs internes en dur
- Utiliser le routing (module/action)
- Préférer les routes nommées (@maroute)
 - Permet de changer
 - D'URL : pour diffuser des URL lisibles adaptables sans changer la logique
 - D'URI : afin de changer une action de module pour un refactoring



Maintenance : utiliser le routing (2/3)



BONNE PRATIQUE 19

Routing propre : faire des routes nommées

URL

contenu/editorial-du-10-juin

Routing

@article

URI

Module : article

Action : show

Paramètres:

Titre = editorial-du-10-juin

- Permet de changer les URL
- Maintenance interne aisée :
 - déplacer du code de module



Maintenance : utiliser le routing (3/3)



BONNE PRATIQUE 19

Et du code vers l'extérieur ?

```
<?php echo link_to('@article?title=editorial-du-10-juin') ?>
```

Devient bien:

```
contenu/editorial-du-10-juin
```

Stable aux changements en bidirectionnel !



Maintenance : environnements



BONNE PRATIQUE 12

- Le minimum est d'avoir trois environnements
 - Développement
 - Production
 - Test
- Si le cache est utilisé, le minimum est quatre environnements
 - Développement
 - Staging
 - Production
 - Test



Maintenance : déploiement



BONNE PRATIQUE 20

- Si besoins simples
 - Utiliser les outils de déploiement proposés par Symfony
 - Rsync, pas de FTP
 - Supprimer les contrôleurs de développement
 - Utiliser la ligne de commande
 - Utiliser les tâche symfony de déploiement
 - Permet de mettre à jour un environnement
- Besoin de plus complexe ? Write your own
- Importance de
 - `app:enable/disable`
 - Ne plus faire de FTP



Maintenance : de l'explicite !

BONNE PRATIQUE 21



Préférer un mode explicite

- Actions
 - Pre/post Execute() éviter : préférer un appel direct à méthode protégée de la classe d'actions
- Comportements du modèle
 - Permet un comportement transversal hors héritage
 - Les objets perdent beaucoup de leur caractère prédictible



Les bonnes pratiques



Performances



Les bonnes pratiques du développement symfony en 30 points clés
Marc Hugon, Gilles Taupenas



Performances : accélérateur PHP



BONNE PRATIQUE 22

- Le parsing des fichiers PHP est coûteux
- Installer un accélérateur PHP = gain de performance serveur
 - Xcache (<http://xcache.lighttpd.net/>)
 - APC (<http://php.net/apc>)
- Peut s'installer sans contrainte sur un environnement de développement



Performances : debug tools



BONNE PRATIQUE 23

- Que regarder pour vérifier les performances de l'application ?



- Empreinte mémoire (28223.3KB) : peu pertinent
- Vitesse mesurée (5895ms) : peu pertinent
- Nombre de requêtes : peu pertinent
- La variation et le détail des requêtes sont importantes



Performances : debug tools

BONNE PRATIQUE 23



- Bon réglage des niveaux de log par environnement permet d'aller jusqu'au détail de la web debug toolbar



Performances : cache

BONNE PRATIQUE 24



- Objet
 - Si le résultat d'un calcul / appel en lui-même est important
- HTML
 - S'il est pertinent d'avoir des versions préparées de pages / morceaux de pages
 - Composant, template avec/sans layout
 - Supercache : nécessite authentification ?
 - Ruser : PHP peut rester, Javascript peut compenser
- Taille du cache
 - Bien choisir son identifiant de cache



Performances : cache

BONNE PRATIQUE 24



- Connaître
 - son niveau d'exigence
 - Quand optimiser dans le cycle de développement
- Attention à l'invalidation

Choix

- Local
- Partagé

Spécificités à prendre en compte suivant le type choisi

- Ecrire du memcache / fichier
- Invalider du memcache / fichier



Les bonnes pratiques



D'autres bonnes pratiques



Les bonnes pratiques du développement symfony en 30 points clés
Marc Hugon, Gilles Taupenas



Général : encodage

BONNE PRATIQUE 25



- Conseillé : UTF-8
 - Seul format permettant d’implémenter le multilinguisme
 - Le plus standard pour mettre en place et utiliser des API externes
- Dans tous les cas : s’assurer que l’encodage est le même à tous les niveaux :
 - Fichiers sources
 - Base de données
 - Encodage HTML





Côté templates

- Bon découpage
- Pratique courante chez Sensio : intégrer le helper `i18n` sur le vocable présent dans les templates

⇒ Surcôté léger de développement

- Utiliser le helper `format_number_choice()`
- Les termes doivent rester côté vue



Internationalisation

BONNE PRATIQUE 25



Côté contenu utilisateur à adapter (stocké en base)

- Attention aux jointures
- Attention au comportement avec les behaviors





- Utilisation directe de `$_GET`, `$_POST`?
- Désactiver l'échappement symfony :
 - Localement :

```
$obj->getXXX(ESC_RAW)
```

```
$obj->getRawValue()
```
 - Globalement :

```
Escaping_strategy : off
```
- Attaques très aisées : CSRF, XSS



Pratique : développer en plugins



BONNE PRATIQUE 27

- Ne pas hésiter à utiliser des plugins spécifiques à l'application
- Facilite l'intervention de plusieurs développeurs simultanément
- Facilite la réutilisation entre plusieurs projets, mais ce n'est pas un objectif en soi



Pratique : plugins externes



BONNE PRATIQUE 28

- Privilégier les plugins existants quand c'est possible :
 - Inutile de réécrire ce qui existe déjà
- S'assurer de la « qualité » du plugin
 - Ouï dire
 - Indicateur de nombre d'utilisation sur le site symfony-project.org
 - Lire le code
- Ne pas faire d'export sur le trunk



Pratique : versioning

BONNE PRATIQUE 29



- Le choix vous appartient (SVN, CVS, GIT)
- Avantages
 - Simplifie le travail collaboratif
 - Historique
 - Retour arrière possible
- Bonnes pratiques
 - Commenter tout modification
 - Un « commit » quotidien ?





- SVN
 - Ignore
 - External : installation conseillée de symfony

- Taguer les versions
 - Déployées
 - En prod
 - En preprod
 - Pour le suivi et la maintenance des versions



Pratique : Gestion des droits d'accès

BONNE PRATIQUE 30



- Ne pas confondre credential (droit) et un profil utilisateur
- Credential : connotation minimaliste, à utiliser pour des cas simples (un front office avec une zone réservée aux utilisateurs authentifiés)
- Profil utilisateur : comportements différents de l'application selon des droits



Les bonnes pratiques



Une dernière, très importante

Ne pas faire faire aux applications autre chose que de l'applicatif !

- Pas d'architecture physique !
- Pas de synchronisation multi-points
- ...
- Stop !



Les bonnes pratiques



Merci et....

Questions ?

