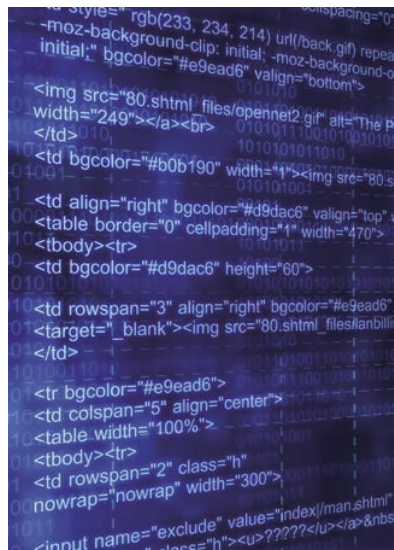




Allons plus loin dans Symfony

Jonathan Démoutiez

Symfony est un framework développé en PHP5. Inspiré de Ruby On Rails, Symfony apporte au développeur les outils nécessaires pour développer plus rapidement et proprement, que ce soit des sites ou des applications. Bien que très récent, il est déjà dominant face à ses concurrents et suscite énormément d'intérêt et de curiosité.



linux@software.com.pl

Symfony est un framework Open Source développé en PHP5. Inspiré de Ruby On Rail, il intègre *PROPEL* (O.R.M.), *CREOLE* (Couche d'abstraction de base de données) et *Mojavi* (implémentation M.V.C.).

En réutilisant l'existant au lieu de repartir à zéro, Symfony reste à ce jour l'une des solutions les plus complètes en PHP.

Un projet Symfony se fonde sur la génération d'applications qui permettra de séparer, par exemple, le front-office et le back-office. Une application se découpe ensuite en modules qui permettront de séparer chaque fonctionnalité d'une application (par exemple : inscription, espace membre, catalogue, paiement...).

Ce principe nous permet d'obtenir un projet clair et structuré.

Environnement

Symfony distingue deux environnements de travail : le premier est un environnement de développement `define(SF_ENVIRONMENT', 'dev');` et le second concerne la production `define('SF_ENVIRONMENT',`

`'prod');` (en plus de l'environnement de test). Ces derniers permettront de donner un comportement différent à notre application : affichage des erreurs et warnings, gestion ou non du cache, gestion des différents accès à la base de données...

L'environnement de développement nous permettra donc de debugger notre application grâce aux outils fournis, en affichant clairement les erreurs avec un tracking (plutôt qu'une page bateau *error 500* en



Ce qu'il faut savoir...

- Les notions de PHP5,
- Les bases du framework Symfony.



Cet article explique...

- Symfony à un niveau intermédiaire,
- Ses nombreuses possibilités,
- La souplesse du framework.



environnement de production). L'outil de débogage *sfDebug*, activé via l'instruction `define('SF_DEBUG', 1);` génère une *toolbar* en haut à droite de la fenêtre. Il nous donne de nombreuses informations utiles (temps de génération de la page, toutes les requêtes sql exécutées, les variables passées en paramètre / en session...). Cette toolbar, bien que très pratique, a cependant ses limites : elle ne se met pas à jour lors, par exemple, d'appels en Ajax.

Configuration – YAML

Remarquez que pour la configuration d'un projet, Symfony a opté pour des fichiers de configuration *YAML*.

Remarquez que dans le même esprit que *XML*, l'idée est que toutes les données peuvent se représenter par un assemblage de tableaux.

Ainsi Symfony inclut un système complet et très efficace de configuration via des fichiers *YAML* (extension : *yml*). Détaillons ci-dessous les principaux fichiers de configuration d'une application.

view.yml

Le fichier *view.yml* sert à configurer le layout (la vue générale) de l'application (Listing 1) :

- balises *meta*,
- les fichiers CSS à inclure,
- les fichiers JS à inclure,
- la template à appeler.

Le Listing 1 montre comment définir :

- les balises *meta* souhaitées,
- les feuilles de style à utiliser (*/css/main.css, /css/style.css*),
- les fichiers *javascript* à appeler (*/js/linux_plus.js*),
- l'utilisation d'un layout *layout* : (*apps/linux_plus/templates/layout_linux_plus.php*).

routing.yml

Ce fichier nous permet de décrire très simplement des règles de réécriture d'URL grâce au système d'URI de Symfony (Listing 2).

Le Listing 2 décrit, en plus des règles de réécriture par défaut, une règle *fiche_article*. Cette règle va nous permettre d'optimiser nos URL pour le référencement. Au lieu d'avoir l'url : *http://domaine/Article/index/article_id/1* nous obtiendrons : *http://domaine/article/1/mon-premier-article*.

settings.yml

Grâce à ce fichier, configurerons de très nombreuses choses pour notre application, tout en différenciant l'environnement dans lequel nous sommes (DEV ou PROD). Je vous renvoie au fichier de configuration déjà entièrement commenté.

security.yml

Ce fichier nous aidera à gérer les droits d'accès en un temps record (Listing 3).

Le Listing 3 propose un exemple de configuration d'accès. Ici, l'utilisateur devra être authentifié et posséder le *credential* admin (parfait pour le back-office par exemple).

filters.yml

Symfony intègre un système de filtres très efficace, permettant entre autres, de sécuriser l'application mais aussi de modifier le traitement d'une action selon certaines données (Listing 4). Par exemple, une *black-list* d'adresses IP pourrait très bien être gérée via un filtre. Les filtres s'exécutent en cascade. Arrivés au traitement de l'action, ils s'exécutent dans le sens inverse (Figure 1).

Le Listing 4 montre comment insérer nos propres filtres. Ici, entre le filtre de sécurité et celui du cache, j'ai intégré mon filtre de *black-list* d'adresses IP.

Développer les vues

Symfony utilise une architecture M.V.C., ce qui signifie que notre projet sera structuré en séparant les trois couches (*Modèle – Vue – Controller*). En plus de séparer notre code HTML du reste, un système de *partials* permet de découper nos vues en petites parties afin d'éviter de répéter notre code HTML.

Chaque action est associée à une vue (l'action *index* est associée à la vue *indexSuccess.php*). Une vue peut ensuite inclure des *partials* et / ou utiliser des *Helpers* afin de ne dupliquer aucune ligne de code.

Il est très important de bien penser l'écriture d'un *partial* et des fonctions d'un *Helper*, pour que le tout reste compatible avec n'importe quel appel. Ceci dans le but d'éviter la génération de bugs en cascade. Aussi, pensez à écrire vos *Helpers* en gardant en tête son éventuelle utilisation dans un autre projet.

Lorsque nous avons une application bien découpée et bien pensée (DRY : *Don't repeat yourself*), nous avons une application très souple

en terme d'évolution, de performance et de maintenance (tout l'atout de ce framework).

Les helpers

Un helper est un fichier qui contient des fonctions. Ces fonctions nous permettront de générer du

Listing 1. Exemple du fichier view.yml

```
default:
  http metas:
    content-type: text/html
  metas:
    title: LinuxPlus application
    description: LinuxPlus
  application
    keywords: linux, plus, dvd
    language: fr
  stylesheets: [main, style]
  javascripts: [linux_plus]
  has_layout: on
  layout: layout_linux_plus
```

Listing 2. Exemple du fichier routing.yml

```
fiche_article:
  url: /article/:article_id/:
  article_nom
  param: { module: Article,
  action: index }
# default rules
homepage:
  url: /
  param: { module: Home, action:
  index }
default_symfony:
  url: /symfony/:action/*
  param: { module: default }
default_index:
  url: /:module
  param: { action: index }
default:
  url: /:module/:action/*
```

Listing 3. Exemple du fichier security.yml

```
default:
  is_secure: on
  credentials: admin
```

Listing 4. Exemple du fichier filters.yml

```
rendering: ~
web_debug: ~
security: ~
blackListIPFilter:
  class: BlackListIPFilter
cache: ~
common: ~
flash: ~
execution: ~
```



code HTML ou Javascript. Symfony contient des helpers qui vous simplifieront la vie. Par défaut, trois helpers sont chargés dans l'application :

- partial (gestion des partiels),
- cache (gestion du cache),
- form (gestion des formulaires).

D'autres partiels sont généralement très utilisés : Date et DateForm, I18N, Javascript, Validation (gestion des erreurs de formulaire).



À savoir

- Chaque fichier de configuration d'une application peut être dupliqué dans le dossier *config* d'un module. Ainsi, chaque module spécifie sa propre configuration. Exemple : par défaut, pour accéder à l'application, l'utilisateur n'a pas besoin d'être authentifié, mais nous voulons sécuriser le module `mon_compte`. Dans ce cas, nous le précisons via le fichier *security.yml* que nous créons dans le dossier *config* de ce module,
- Le paramètre `OPTIONS_HTML` est un tableau qui permet de spécifier les attributs de nos balises HTML. Par exemple, pour générer un lien avec l'attribut `title='mon lien'`, nous passons en paramètre `:array('title' => 'mon lien')`.

Listing 5. Génération de lien HTML

```
link_to(TEXTE, URI, OPTIONS_HTML); // Génère un lien vers une autre page,
link_to_function(TEXTE, FONCTIONS, OPTIONS_HTML); // Génère un lien vers
// des instructions Javascript,
link_to_remote(TEXTE, OPTIONS, OPTIONS_HTML); // Génère un lien vers
// un appel en AJAX,
link_to_if(CONDITIONS, TEXTE, URI, OPTIONS_HTML); // Affiche le texte sans
// le lien si les conditions ne sont pas remplies.
```

Listing 6. Les formulaires

```
input_tag(NOM, VALEUR, OPTIONS_HTML);
input_date_tag(NOM, VALEUR, OPTIONS_HTML);
input_password_tag(NOM, VALEUR, OPTIONS_HTML);
input_hidden_tag(NOM, VALEUR, OPTIONS_HTML);
checkbox_tag(NOM, VALEUR, CONDITIONS, OPTIONS_HTML);
textarea_tag(NOM, VALEUR, OPTIONS_HTML);
select_tag(NOM, VALEURS, OPTIONS_HTML);
```

Listing 7. Exemple de schéma

```
categorie_article:
  id:
categorie_article_i18n:
  id:
  nom: varchar(255)
article:
  id:
  created_at:
  updated_at:
  categorie_article_id:
  article_i18n:
    titre: varchar(255)
    texte: longvarchar
```

Listing 8. Exemple de Criterion

```
$criterion1 = $c->getNewCriterion(ArticlePeer::CATEGORIE_ID,
    $categorie->getId());
$criterion2 = $c->getNewCriterion(ArticlePeer::CATEGORIE_ID,
    $sous_categorie->getId());
$criterion1->addOr($criterion2);
$c->add($criterion1);
```

Les fonctions les plus utilisées

Dans cette partie, nous verrons les nombreuses fonctions de Symfony que nous utiliserons en permanence dans nos templates.

Seule particularité, les valeurs de la balise `SELECT`. Soit nous passons en paramètre une chaîne de caractères avec nos options par exemple :

```
<option value='1'>Valeur 1</option>
```

Soit nous passons un tableau via la fonction `options_for_select` par exemple :

```
options_for_select(array('1' =>
    'Valeur 1'), 'VALEUR PAR
    DEFAULT')
```

Sinon, il n'y a rien de particulier pour ces fonctions contenues dans l'helper *Form*, leurs noms parlent d'eux-même.

Base de données

Symfony utilise *propel* un O.R.M. pour PHP 5 qui permet de créer une correspondance entre le monde objet du langage et le monde relationnel de la base de données. Chaque table se verra associer deux classes, une pour gérer les données/instances d'objet, l'autre (entièrement statique) pour gérer le lien avec la base de données (ainsi, pour une table `article`, nous aurons les classes `Article` et `ArticlePeer`). La génération de ces classes se fait via un schéma de configuration YAML (et oui, encore eux). Si vous partez de zéro, il est préférable d'écrire le schéma en suivant les conventions détaillées juste après (Listing 5) puis de générer la base à partir de lui `symfony propel-build-all` (cette commande réinitialise votre base de données). Sinon, générez-le à partir de votre base existante `symfony propel-build-schema` (puis éventuellement modifiez-le).

Une fois votre schéma décrit, générez les classes objet (le modèle) par la commande `symfony propel-build-model`. Cette dernière n'effacera ni la classe `Article` ni la classe `ArticlePeer`, mais ré-initialisera les classes `BaseArticle` et `BaseArticlePeer` dont elles héritent respectivement (c'est pour cela qu'il ne faut pas les modifier).

Conventions

Si votre structure de données adopte les conventions de l'O.R.M. l'écriture du schéma n'en sera que plus souple. Les noms de champs :



Listing 9. Utilisation de Criteria

```
public function executeIndex() {
    $this->setVar('articles', ArticlePeer::doSelect(new Criteria()));
}
// indexSuccess.php

<?php foreach ($articles as $article): ?>
    <p class='Titre'>
        <?php echo $article->getTitre() ?>
    </p>
    <p class="Titre2" align="center">
        <i><?php echo __("Publié le") ?> : <?php echo format_date($article
            ->getCreatedAt(), 'P') . __(" à ") . format_date($article
            ->getCreatedAt(), 't') ?></i>
    </p>
    <p class='article'>
        <?php echo $article->getTexte() ?><br />
    </p>
<?php endforeach ?>
```

- id, automatiquement considéré comme un entier et comme la clé primaire,
- created_at, automatiquement considéré comme un champ de type *datetime*, cette donnée se remplit automatiquement à la première sauvegarde d'une instance,
- updated_at, automatiquement considéré comme un champ de type *datetime*, cette donnée se met à jour à chaque sauvegarde d'une instance,
- nom_table_id, automatiquement considéré comme un entier et comme la clé étrangère de la table nom_table (très pratique pour notre système relationnel).

Les noms de tables :

- nom_table_i18n, automatiquement considéré comme la table complémentaire en cas de multilinguisme (lors de la génération de la table, le champ

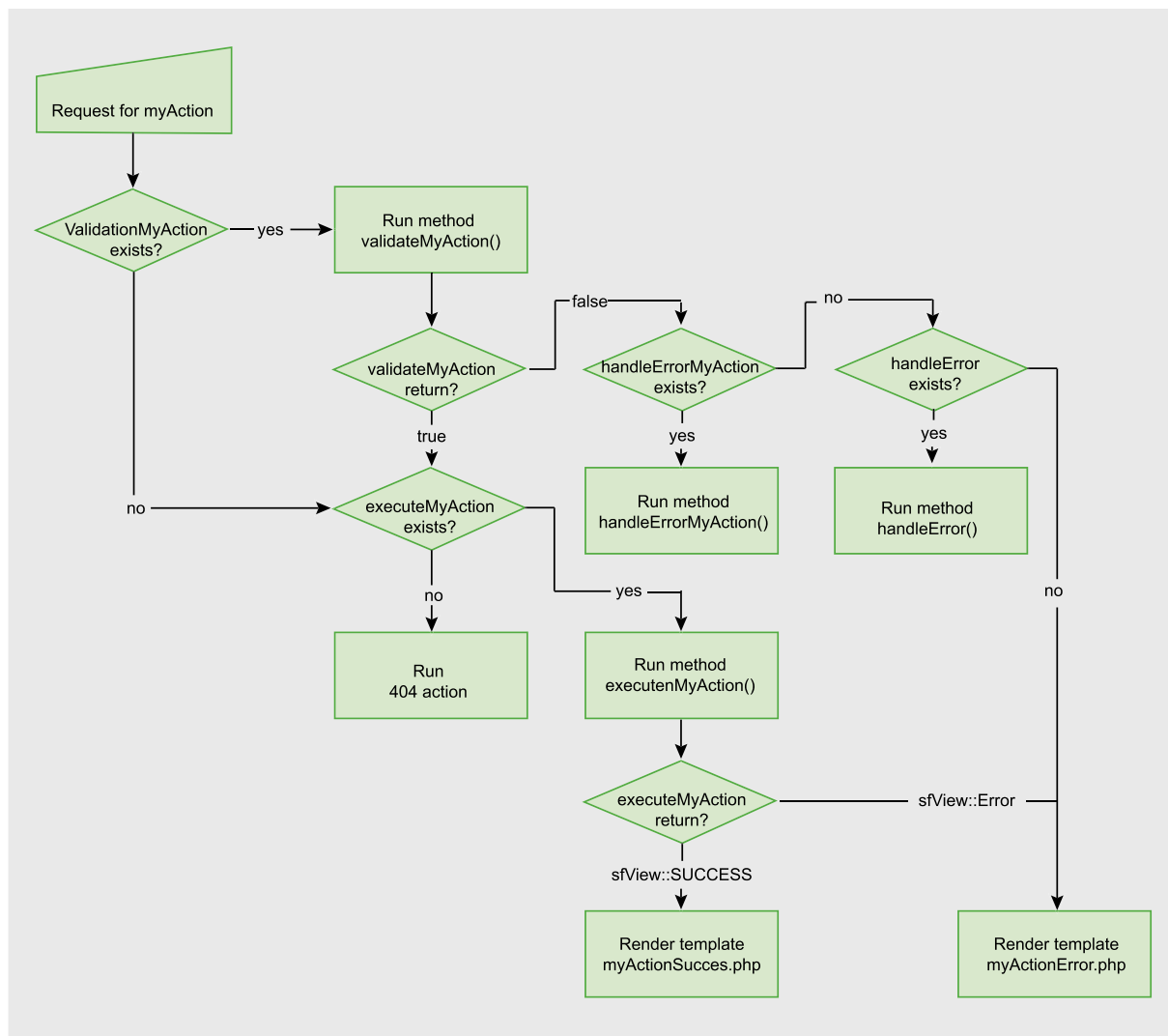


Figure 1. Système de validation d'action



culture est automatiquement généré sur cette table.

L'Object-Relational Mapping (O.R.M.)

`Criteria` est une classe liée à Propel qui nous permettra d'écrire nos requêtes sous forme d'instance d'objet. Nous n'aurons donc plus à écrire nos requêtes en SQL (sauf en cas complexe ou d'optimisation ou il sera toujours possible de revenir à la méthode à l'ancienne).

Pour la suite de cette partie, admettons le schéma où nous pourrons créer toute forme de requêtes en les écrivant comme une instance de la classe `Criteria`. Pour les requêtes les plus simples, voici comment procéder : créez une instance, `$c = new Criteria();`, ajoutez vos critères, `$c->add(ArticlePeer::TITRE, '%test%', Criteria::LIKE);`, récupérez le résultat, `$articles = ArticlesPeer::doSelect($c);`.

Le troisième paramètre de la méthode `add`, est optionnel. Par défaut, l'opérateur `=` est utilisé. Je vous renvoie à la documentation officielle pour voir la liste des correspondances des différents opérateurs.

Toutefois, la méthode `add` n'autorise que l'ajout des conditions à réaliser (séparées par l'opérateur `AND` de SQL).

Pour utiliser l'opérateur `OR`, il faudra passer par des *Criteria*, (Listing 6). Récupérez maintenant vos articles dans une action pour les afficher dans votre vue (Listing 7).

L'utilisation de la méthode `__()` permet d'implémenter le multilinguisme que nous verrons plus loin.

Symfony connaît la relation entre la table `catégories` et la table `articles` grâce aux informations du schéma de données. Ainsi, toutes les méthodes utiles sont déjà à notre disposition. À partir d'un article, les méthodes `Categorie::getCategorie()` et

`setCategorie(Categorie $categorie)` permettent de récupérer ou de modifier la catégorie associée (via instance d'objet). Dans l'autre sens, la table `catégories` nous fournit les méthodes, `addArticle(Article $article)`, `Article::getArticles()` et `int countArticles()` (toujours via instance d'objet).

L'implémentation objet du système relationnel est plus que pratique, il nous devient par la suite très difficile de s'en passer.

Validation d'action

Au même titre que la méthode `executeIndex`, il existe des méthodes qui permettent de couper notre action en plusieurs couches :

Listing 10. Validation d'action

```
public function handleErrorIndex() {
    $this->redirect('/');
} public function validateIndex() {
    $this->setVar('article', ArticlePeer::retrieveByPk($this->getRequestParameter('id')));
    return $this->article;
} public function executeIndex() {
    // Pas de traitement particulier dans l'action
}
```

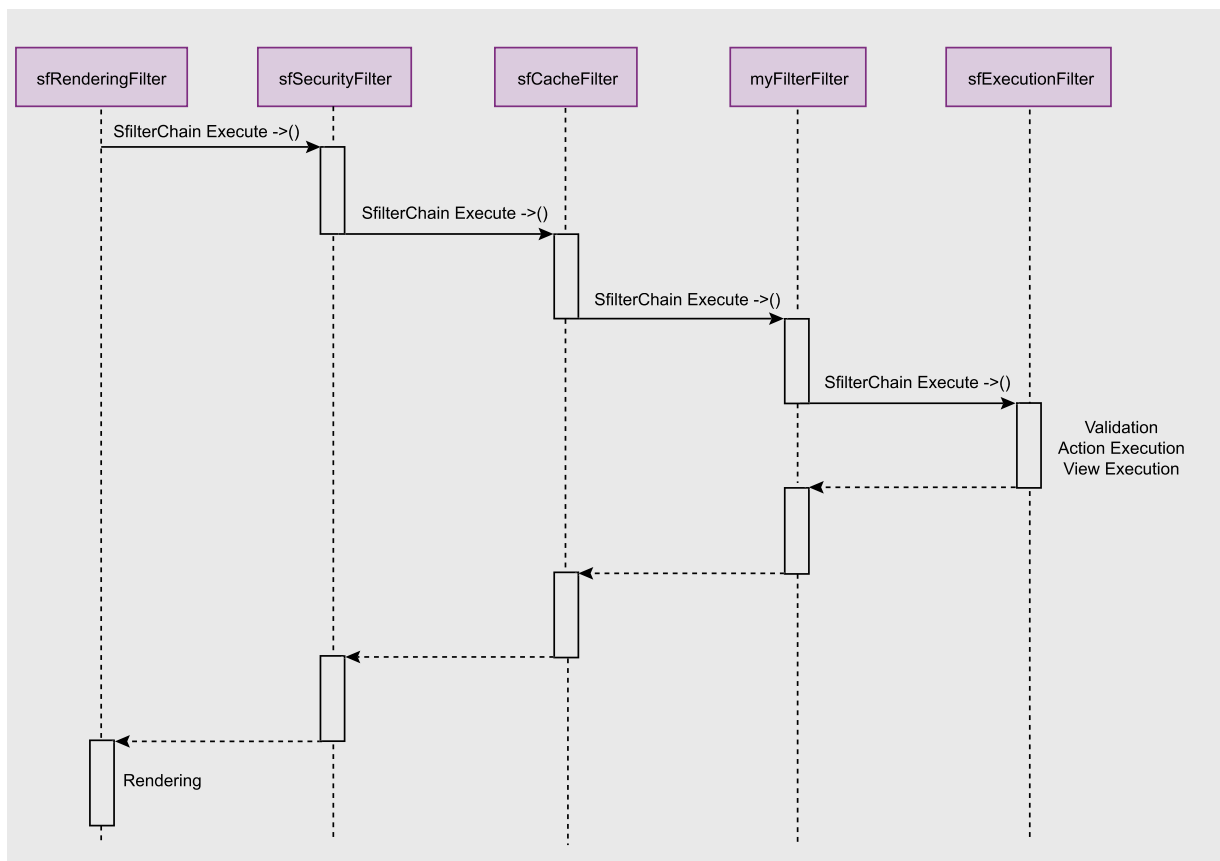


Figure 2. Exemple d'une chaîne de filtre



validation des données / de l'action, exécution de l'erreur / exécution de l'action. La méthode `validateIndex` doit retourner un booléen. Si elle retourne `false` la méthode `executeIndex` ne sera pas exécutée. A défaut de la méthode `handleErrorIndex` qui, après son exécution, appellera non plus la template `indexSuccess.php` mais `indexError.php`. ATTENTION si vous oubliez de retourner votre booléen dans la méthode `validate`, elle retournera automatiquement `false` (Figure 2 et Listing 8).

Une autre couche de validation existe lors de l'utilisation de formulaire via le fichier de configuration YAML. Ce fichier se place dans le dossier `validate` du module concerné et se nomme `: nom_action.yml`. Dans ce fichier, nous pourrions spécifier très simplement de nombreuses vérifications : champs obligatoires, champ unique, de type entier, compris entre telle et telle valeur, de telle longueur... (Listing 9).

D'autres *validators* sont disponibles : `sfStringValidator`, `sfNumberValidator`, `sfEmailValidator`, `sfUrlValidator`, `sfRegexValidator`, `sfCompareValidator`, `sfPropelUniqueValidator`, `sfFileValidator`, `sfCallbackValidator`. Bien sûr, vous pouvez également créer les vôtres.

Le multilingue

Le développement d'une application multilingue est parfois cauchemardesque, mais grâce Symfony, cela devient une partie de plaisir !

Commençons par activer le système dans notre application : activons le mode `i18n` dans le fichier de configuration `settings.yml` en décommentant la ligne correspondante (désolé il n'y avait pas plus simple). Puis, configurons le système via le fichier `i18n.yml` (parmi les autres fichiers de configuration) (Listing 10). Il n'y a rien de spécifique pour ce fichier, juste à préciser que XLIFF correspond à un dictionnaire de traduction (*XML Localization Interchange File Format*).

Notre application est maintenant prête pour fonctionner en multilingue. Dans notre exemple, la culture par défaut est le français, mais Symfony se réfère toujours à la culture en session de l'utilisateur. Pour la modifier dans un contrôleur, rien de plus simple : `$this->getUser()->setCulture('en');`

Il existe deux types de données à traduire. Tout d'abord, le texte est directement dans notre code HTML. Pour cela, au lieu d'écrire notre texte à nu, nous passons par la méthode `<?php echo __('MON TEXTE A TRADUIRE') ?>`. Par la suite, il ne

nous reste plus qu'à définir nos traductions dans un dictionnaire pour l'anglais vers le français `messages.fr.xml` et à placer dans le dossier `i18n` à la racine de l'application (Listing 11).

Il existe également des textes stockés en base (référez-vous à l'explication précédente pour la structure des tables). Pour que l'O.R.M. récupère les données en fonction de la culture de l'utilisateur, il suffit de redéfinir la méthode `hydrate` dans les classes

correspondantes, en reprenant l'exemple de notre structure précédente, dans la classe `Article`.

Notre application est déjà entièrement multilingue, efficace non ?

Le cache

Il est fréquent d'avoir des parties dynamiques restant statiques à cours ou moyen terme. Il s'agit en général de données qui sont rarement modifiées, comme une liste de

Listing 11. Validation d'un formulaire

```
methods: [post]
fillin:
    enabled: true
fields:
    pseudo:
        required: yes
        sfPropelUniqueValidator:
            class: User
            column: pseudo
            unique_error: Ce pseudo est déjà utilisé
    password1:
        required: yes
    password2:
        required: yes
        sfCompareValidator:
            check: password1
            compare_error: Les deux mots de passe ne correspondent pas.
    email:
        required: yes
        sfEmailValidator:
            strict: true
            email_error: L'adresse email est invalide.
```

Listing 12. Exemple de dictionnaire de traduction

```
<?xml version="1.0" ?>
<xliff version="1.0">
    <file original="global" source-language="fr_FR" datatype="plaintext">
        <body>
            <trans-unit id="1">
                <source>no result</source>
                <target>aucun résultat</target>
            </trans-unit>
            <trans-unit id="2">
                <source>create</source>
                <target>créer</target>
            </trans-unit>
        </body>
    </file>
</xliff>
```

Listing 13. Redéfinition de la méthode hydrate

```
public function hydrate(ResultSet $rs, $startcol = 1) {
    parent::hydrate($rs, $startcol);
    $this->setCulture(sfContext::getInstance()->getUser()->getCulture());
}
```



catégories par exemple. Si nous avons une structure récursive qui nous permet de gérer des sous-catégories, la récupération des données peut nécessiter un certain nombre de requêtes. Dans ce genre de cas, il est très intéressant de mettre le résultat dans le cache de l'application. Ceci afin de minimiser le nombre de requêtes et d'optimiser le temps de réponse. Pour cela, la méthode la plus simple consiste à développer le code dans un partial et de mettre en cache les partiels voulus en modifiant le fichier de configuration `cache.yml` (Listing 12).

Le `_` devant les noms signifie qu'il s'agit de partiels. Sans ce caractère, Symfony considère que nous parlons de nom d'action.

À ce niveau, vous devez commencer à réellement vous rendre compte de l'efficacité et de la productivité du framework, n'est ce pas ? Pourtant, c'est loin d'être fini...

L'admin generator

Combien de temps vous faut-il pour développer un système CRUD de vos différentes tables ? Listing des données, création/édition/suppression, le développement des formulaires, la récupération des données, le traitement... 4 heures ? 1 journée ? 3 journées ???? et si je vous réponds 10 minutes ? Oui, sérieusement.

Créons une application pour notre back-office symfony init-app back. Il ne reste plus qu'à générer un module pour chaque table

voulue `symfony propel-init-admin Article Article`. C'est terminé, rendez-vous avec votre navigateur dans le module Article. Certes cela reste basique et ce n'est pas exactement ce que vous vouliez : tous les champs sont pris en compte et pas forcément comme il faut...

Passons à la configuration. Éditez le fichier `app/back/modules/Articles/config/generator.yml` (Listing 13). Nous pourrions définir les champs à afficher et préciser le type de certains champs spécifiques.

Dans cet exemple, nous obtiendrons directement un éditeur de texte (ici tinymce) sur notre champ texte en précisant la largeur de la *boite*. le `=` devant le nom du champ signifie qu'il s'agit d'un champ relationnel et que nous nous fierons à la table correspondante et à la méthode `__toString` (à redéfinir) de la classe (nous obtiendrons une `select_tag`). Il est possible de rajouter des spécificités avec le préfixe `_` pour faire appel à un partial ou `~` pour un composant.

La méthode `peer` utilisée par défaut est la méthode `doSelect`. Ici, je précise `doSelectJoinAll` afin d'éviter de nombreuses requêtes inutiles.



Sur Internet

- Le site officiel de Symfony : <http://www.symfony-project.org/>,
- Le groupe google en français : <http://groups.google.com/group/symfony-fr>,
- Le blog de l'auteur de cet article regroupant notamment une documentation sur Symfony : <http://jonathan.demoutiez.net/>.



Terminologie

- *propel* est un O.R.M. (*mapping objet-relationnel*), il permet de faire une correspondance entre le monde objet du langage et le monde relationnel de la base de données,
- *creole* est une couche d'abstraction de base de données, elle permet de gérer n'importe quel gestionnaire de base de données,
- *m.v.c.* : Le principe M.V.C. permet de séparer les couches `MODELES` (Base de données), `VUES`, `CONTROLEURS`.

Listing 14. Exemple du fichier generator.yml

```
generator:
  class: sfPropelAdminGenerator
  param:
    model_class: Article
    theme: default
    list:
      title: Liste des articles
      sort: [id, desc]
      display: [id, created_at, titre, texte, =categorie_article_id]
      max_per_page: 10
      object_actions:
        _delete: ~
        _edit: ~
    edit:
      title: Création/Édition d'un article
      peer: doSelectJoinAll
    fields:
      texte: { type: textarea_tag, params: rich= tinymce tinymce_
        options=width:800 }
      display: [id, created_at, culture, titre, texte, =categorie_
        article_id]
```

Conclusion

Les auteurs de Symfony se sont très largement inspirés du framework Ruby on Rails pour créer un cadre de travail puissant et organisé en PHP.

Apportant principalement une séparation des couches selon le modèle MVC, une couche de *mapping objet-relationnel* (O.R.M.) et un support Ajax en plus de toute la panoplie que nous avons vue dans cet article et de ce que vous découvrirez tout au long de votre apprentissage sur cet énorme framework, Symfony est le plus grand framework PHP du moment.

Mais les auteurs n'en restent pas là. Symfony ne cesse d'évoluer notamment à travers les plugins développés par la communauté, mais aussi par les auteurs eux-même, actuellement sur la version 1.0. La version 1.1 devrait arriver très prochainement et les suivantes sont déjà prévues. 🐘



A propos de l'auteur

Jonathan Démoutiez est actuellement responsable technique chez Webpulsar, un studio de développement web 2.0 basé sur Lille. Passionné d'informatique, il évolue depuis huit ans dans le monde de PHP et plus récemment, dans le monde de Ruby On Rails. Site de l'auteur : <http://jonathan.demoutiez.net>.