[Partie 5] - Personnalisation de la vue : extensions Twig, barre latérale et Assetic

Introduction

Dans ce chapitre, nous allons continuer à construire la partie utilisateur de Symblog. Nous allons améliore la page d'accueil pour aficher des informations sur les commentaires associés aux articles, ainsi qu'améliorer les résultats potentiels de recherche via SEO (Search Engine Optimization : optimisation pour les moteurs de recherches) en ajoutant le titre des articles dans l'URL. Nous allons également commencer à travailler sur la barre latérale, en lui ajoutant 2 composants classiques; un nuage de tags et une section "Derniers commentaires". Nous allons également explorer les différents environnements dans Symfony2 et apprendre comment lancer Symblog dans l'environnement de production. Le moteur de template Twig va être étendu afin de proposer un nouveau filtre, et nous allons présenter Assetic pour la gestion des ressources externes.

La page d'accueil - Articles et commentaires

Pour le moment, la page d'accueil se contente d'afficher les articles mais ne fournit pas d'informations concernant les commentaires qui leurs sont associés. Maintenant que nous avons construit une entité `Comment`, nous pouvons mettre à jour la page d'accueil afin d'ajouter ces informations. Comme nous avons déjà établi un lien entre les entités `Blog` et `Comment`, nous savons que Doctrine 2 est capable de retrouver les commentaires associés à un article (souvenez vous que nous avons ajouté un membre `$comments` dans l'entité `Blog`). Mettons à jour le template de la page d'accueil dans `src/Blogger/BlogBundle/Resources/views/Page/index.html.twig` avec ce qui suit.

```
{# src/Blogger/BlogBundle/Resources/views/Page/index.html.twig #}

{# .. #}

<footer class="meta">
    <p>Comments: {{ blog.comments|length }}</p>
    <p>Posted by <span class="highlight">{{ blog.author }}</span> at {{ blog.created|date('h:iA') }}</p>
    <p>Tags: <span class="highlight">{{ blog.tags }}</span></p>
</footer>

{# .. #}
```

Nous avons utilisé le getter `comments` afin de récupérer les commentaires de l'article, et avons ensuite passé la liste dans le filtre Twig `length`. Si vous regardez maintenant la page d'accueil via `http://symblog.dev/app_dev.php/`, vous pourrez voir que le nombre de commentaires de chaque article est affiché.

Comme expliqué plus haut, nous avons déjà informé Doctrine 2 que le membre `$comments` de l'entité `Blog` estn associé à l'entité `Comment`. Nous avons réalisé cela dans le chapitre précédent avec la métadonnée suivante dans l'entité `Blog`, dans `src/Blogger/BlogBundle/Entity/Blog.php`.

```
// src/Blogger/BlogBundle/Entity/Blog.php

/**
 * @ORM\OneToMany(targetEntity="Comment", mappedBy="blog")
 */
protected $comments;
```

Nous savons ainsi que Doctrine 2 est conscient de la relation entre articles et commentaires, mais comment a-t-il remplit le membre `$comments` avec les entités `Comment` correspondantes ? Si vous vous souvenez de la méthode que nous avons créé pour le `BlogRepository` (voir ci-dessous), vous pourrez voir que nous n'avons fait aucune sélection des commentaires pour récupérer les articles de la page d'accueil.

```
// src/Blogger/BlogBundle/Repository/BlogRepository.php

public function getLatestBlogs($limit = null)
{
    $qb = $this->createQueryBuilder('b')
               ->select('b')
               ->addOrderBy('b.created', 'DESC');

    if (false === is_null($limit))
        $qb->setMaxResults($limit);

    return $qb->getQuery()
              ->getResult();
}
```

Néanmoins, Doctrine 2 utilise un processus appelé chargement feignant (lazy loading) où les entités `Comment` sont cherchées dans la base de donnée lorsque c'est nécessaire, dans le cas présent lors de l'appel à `{{ blog.comments|length }}`. Nous pouvons démontrer ce processus à l'aide de la abrre d'outils pour développeurs. Nous avons déjà commencé à parler de cet outil, et il est maintenant temps d'aborder l'une des ses fonctionnalités les plus puissantes, le profiler pour Doctrine 2. On se rend dans le profiler Doctrine 2 en cliquant sur le dernier icone de la barre d'outils. Le chiffre à côté indique le nombre de requêtes exécutées sur la base de données pour l'actuelle requête HTTP.

Barre d'outils pour développeurs - Icône Doctrine 2

Si vous cliquez sur l'icône Doctrine 2, des informations sur les requêtes qui ont été exécutées par Doctrine 2 sur la base de données vous seront présentées.

Barre d'outils pour développeurs - Requêtes Doctrine 2

Comme vous pouvez le voir dans la capture d'écran ci-dessus, il y a plusieurs requêtes vers la base de donnée qui sont executées lorsque la page d'accueil est chargée. La seconde requête récupère les articles dans la base de donnée, et est exécutée en réponse à l'appel de la méthode `getLatestBlogs()` de la classe `BlogRepository`. Après cette requête, vous pouvez trouver plusieurs requêtes qui extraient les commentaires depuis la base de donnée, un article à la fois. On peut le voir grâce à `WHERE t0.blog_id = ?` dans chacune des requêtes, où le `?` est remplaté par la valeur du paramètre (l'identifiant de l'article). Chacune de ces requêtes est liée à un appel de `{{ blog.comments }}` dans le template de la page d'accueil. Chaque fois que cette fonction est effectuée, Doctrine 2 va charger, parce que c'est nécessaire ici et pas avant, et donc de manière feignante, les entités `Comment` associées à une entité `Blog`.

Bien que le lazy loading soit très efficace pour récupérer des entités depuis la base de données, ce n'est pas toujours la manière la plus efficace de procéder. Doctrine 2 fournit la possibilité de `joindre` des entités reliées entre elles lorsqu'une requête a lieu sur la base de données. De cette manière, on peut extraire les entité `Blog` et leurs entités `Comment` associées en une seule requête. Mettez à jour le code du `QueryBuilder` de la classe `BlogRepository` dans `src/Blogger/BlogBundle/Repository/BlogRepositoy.php` pour joindre les commentaires.

```
// src/Blogger/BlogBundle/Repository/BlogRepositoy.php

public function getLatestBlogs($limit = null)
{
    $qb = $this->createQueryBuilder('b')
               ->select('b, c')
               ->leftJoin('b.comments', 'c')
               ->addOrderBy('b.created', 'DESC');

    if (false === is_null($limit))
        $qb->setMaxResults($limit);

    return $qb->getQuery()
              ->getResult();
}
```

Si maintenant vous raffraichissez la page d'accueil et allez examiner la sortie de Doctrine 2 dans la barre d'outils, vous allez remarquer que le nombre de requêtes a chuté de manière drastique. Vous pouvez également voir que la table de commentaires a été jointe à la table d'articles.

Le lazy loading et la jonction d'entités qui sont liées sont deux concepts très puissants, mais qui doivent être utilisés correctement. L'équilibre entre les deux doit être trouvé afin de permettre aux applications de fonctionner aussi efficacement que possible. Au premier abord, il simble attrayant de joindre toutes les entités liées afin de ne jamais avoir à faire du lazy loading et à conserver un nombre faible de requêtes vers la base de données. Il est néanmoins important de se souvenir que plus il y a d'informations à aller chercher dans la base de données, plus les traitements à effectuer par Doctrine 2 pour créer les objets associés aux entités sont lourds. Plus de données signifie également plus d'utilisation mémoire par le serveur pour stocker les objets.

Avant d'avancer, faisant un ajout mineur au template de la page d'accueil. Mettez à jour le template de la page d'accueil dans `src/Blogger/BlogBundle/Resources/views/Page/index.html.twig` pour ajouter un lien vers l'affichage des commentaires de l'article.

```
{# src/Blogger/BlogBundle/Resources/views/Page/index.html.twig #}
```

```
{# .. #}

<footer class="meta">
    <p>Comments: <a href="{{ path('BloggerBlogBundle_blog_show', { 'id': blog.id }) }}#comments">{{ blog.comments|length }}</a></p>
    <p>Posted by <span class="highlight">{{ blog.author }}</span> at {{ blog.created|date('h:iA') }}</p>
    <p>Tags: <span class="highlight">{{ blog.tags }}</span></p>
</footer>

{# .. #}
```

La barre latérale.

Actuellement, la barre latérale de Symblog est un peu vide. Nous allons la mettre à jour en lui ajoutant 2 composants, un nuage de tags et une liste des derniers commentaires.

Le nuage de tags

Le nuage de tags montre les tags des articles, les plus populaires ayant plus d'importance visuelle à travers un affichage plus gros. Pour celà, il nous faut un moyen de récupérer tous les articles de tous les articles. Créons de nouvelles méthodes dans la classe `BlogRepository` pour cela. Mettez à jour la classe `BlogRepository` dans `src/Blogger/BlogBundle/Repository/BlogRepository.php` avec ce qui suit.

```
// src/Blogger/BlogBundle/Repository/BlogRepository.php

public function getTags()
{
    $blogTags = $this->createQueryBuilder('b')
                     ->select('b.tags')
                     ->getQuery()
                     ->getResult();

    $tags = array();
    foreach ($blogTags as $blogTag)
    {
        $tags = array_merge(explode(",", $blogTag['tags']), $tags);
    }

    foreach ($tags as &$tag)
    {
        $tag = trim($tag);
    }

    return $tags;
}

public function getTagWeights($tags)
{
    $tagWeights = array();
    if (empty($tags))
        return $tagWeights;

    foreach ($tags as $tag)
    {
        $tagWeights[$tag] = (isset($tagWeights[$tag])) ? $tagWeights[$tag] + 1 : 1;
    }
    // Shuffle the tags
    uksort($tagWeights, function() {
        return rand() > rand();
    });

    $max = max($tagWeights);

    // Max of 5 weights
    $multiplier = ($max > 5) ? 5 / $max : 1;
    foreach ($tagWeights as &$tag)
    {
        $tag = ceil($tag * $multiplier);
    }

    return $tagWeights;
}
```

Comme les tags sont stockés dans la base de donnée au format CSV (comma separated values, c'est à dire que chaque valeur est séparée de la précédente par une virgule), il nous faut un moyen de séparer et de renvoyer le résultat sous la forme d'un tableau. C'est le rôle de `getTags()`. La méthode `getTagWeights()` se sert ensuite du tableau de tafs pour calculer le poids (weight) de chaque tag à partir de son nombre d'occurences dans le tableau. Les tags sont également mélangés afin d'ajouter un peu d'aléatoire à leur affichage.

Maintenant que nous sommes capable de générer un nuage de tags, il faut l'afficher. Créez une nouvelle action dans le `PageController` dans le fichier `src/Blogger/BlogBundle/Controller/PageController.php` pour gérer la barre latérale.

```
// src/Blogger/BlogBundle/Controller/PageController.php

public function sidebarAction()
{
    $em = $this->getDoctrine()
               ->getEntityManager();

    $tags = $em->getRepository('BloggerBlogBundle:Blog')
               ->getTags();

    $tagWeights = $em->getRepository('BloggerBlogBundle:Blog')
                     ->getTagWeights($tags);

    return $this->render('BloggerBlogBundle:Page:sidebar.html.twig', array(
        'tags' => $tagWeights
    ));
}
```

Cette action est très simple, elle utilise les 2 nouvelles méthodes du `BlogRepository` pour générer le nuage de tags, qu'elle passe ensuite en paramètres à la vue. Il nous faut maintenant créer cette vue, dans `src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig`.

```
{# src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig #}

<section class="section">
    <header>
        <h3>Tag Cloud</h3>
    </header>
    <p class="tags">
        {% for tag, weight in tags %}
            <span class="weight-{{ weight }}">{{ tag }}</span>
        {% else %}
            <p>There are no tags</p>
        {% endfor %}
    </p>
</section>
```

Le template est également très simple. Il traverse les différents tags, en leur associant une classe CSS en fonction de leur poids. Dans cette boucle `for` un peu particulière, on accède aux couples `clé/valeur` du tableau avec `tag` pour la clé et `weight` comme valeur. Il existe plusieurs variations de comment utiliser une boucle ``for`` avec Twig disponible dans la ``documentation <http://twig.sensiolabs.org/doc/templates.html#for>`_.

Si vous regardez le principal template du `BloggerBlogBundle` dans `src/Blogger/BlogBundle/Resources/views/layout.html.twig`, vous pourrez remarquer que nous avions placé un élément temporaire pour le bloc de la barre latérale. On peut maintenant le remplacer, en affichant la nouvelle action de la barre latérale. Souvenez vous que la fonction Twig `render` permet d'afficher le contenu d'une action d'un controlleur, dans le cas présent l'action `sidebar` du controlleur `Page`.

```
{# src/Blogger/BlogBundle/Resources/views/layout.html.twig #}

{# .. #}

{% block sidebar %}
    {% render "BloggerBlogBundle:Page:sidebar" %}
{% endblock %}
```

Enfin, ajoutons de la CSS au nuage de tags. Créez la nouvelle feuille de style dans `src/Blogger/BlogBundle/Resources/public/css/sidebar.css`.

```
.sidebar .section { margin-bottom: 20px; }
.sidebar h3 { line-height: 1.2em; font-size: 20px; margin-bottom: 10px; font-weight: normal; background: #eee; padding: 5px;  }
.sidebar p { line-height: 1.5em; margin-bottom: 20px; }
.sidebar ul { list-style: none }
.sidebar ul li { line-height: 1.5em }
.sidebar .small { font-size: 12px; }
.sidebar .comment p { margin-bottom: 5px; }
.sidebar .comment { margin-bottom: 10px; padding-bottom: 10px; }
.sidebar .tags { font-weight: bold; }
.sidebar .tags span { color: #000; font-size: 12px; }
.sidebar .tags .weight-1 { font-size: 12px; }
.sidebar .tags .weight-2 { font-size: 15px; }
.sidebar .tags .weight-3 { font-size: 18px; }
.sidebar .tags .weight-4 { font-size: 21px; }
.sidebar .tags .weight-5 { font-size: 24px; }
```

Comme nous avons ajouté une nouvelle feuille de style, il faut l'inclure. Mettez à jour le template principale du `BloggerBlogBundle` dans `src/Blogger/BlogBundle/Resources/views/layout.html.twig` avec ce qui suit.

```
{# src/Blogger/BlogBundle/Resources/views/layout.html.twig #}

{# .. #}

{% block stylesheets %}
    {{ parent() }}
    <link href="{{ asset('bundles/bloggerblog/css/blog.css') }}" type="text/css" rel="stylesheet" />
    <link href="{{ asset('bundles/bloggerblog/css/sidebar.css') }}" type="text/css" rel="stylesheet" />
{% endblock %}

{# .. #}
```

> **Note**
>
> Si vous n'utilisez pas les liens symboliques pour référencer les fichiers externes dans le répertoire `web`, vous devez relancer la commande suivante afin de copier les nouveaux fichiers CSS.
>
> ```
> $ php app/console assets:install web
> ```

Si vous mettez maintenant à jour la page d'accueil de Symblog, vous verrez que le nuage de tags est affiché dans la barre latérale. Afin que les tags soient affichés avec différents poids, vous devrez modifier les tags factices afin que certains soient plus utilisés que d'autres.

Commentaires récents.

Maintenant que le nuage de tags est en place, ajoutons un composant pour les derniers commentaires à la barre latérale.

Il nous faut tout d'abord un moyen de récupérer les derniers commentaires des articles. Nous allons pour cela ajouter une méthode dans le `CommentRepository` situé dans `src/Blogger/BlogBundle/Repository/CommentRepository.php`.

```
<?php
// src/Blogger/BlogBundle/Repository/CommentRepository.php

public function getLatestComments($limit = 10)
{
    $qb = $this->createQueryBuilder('c')
                ->select('c')
                ->addOrderBy('c.id', 'DESC');

    if (false === is_null($limit))
        $qb->setMaxResults($limit);

    return $qb->getQuery()
              ->getResult();
}
```

Maintenant, mettez à jour l'action de la barre latérale dans `src/Blogger/BlogBundle/Controller/PageController.php` afin de récupérer les derniers commentaires et les fournir à la vue.

```
// src/Blogger/BlogBundle/Controller/PageController.php

public function sidebarAction()
{
    // ..

    $commentLimit   = $this->container
                           ->getParameter('blogger_blog.comments.latest_comment_limit');
    $latestComments = $em->getRepository('BloggerBlogBundle:Comment')
                         ->getLatestComments($commentLimit);

    return $this->render('BloggerBlogBundle:Page:sidebar.html.twig', array(
        'latestComments'    => $latestComments,
        'tags'              => $tagWeights
    ));
}
```

Vous remarquerez également que nous avons utilisé un nouveau paramètre appelé `blogger_blog.comments.latest_comment_limit` afin de limiter le nombre de commentaires à afficher. Pour créer ce paramètre, mettez à jour le fichier de configuration dans

`src/Blogger/BlogBundle/Resources/config/config.yml` avec ce qui suit.

```
# src/Blogger/BlogBundle/Resources/config/config.yml

parameters:
    # ..

    # Blogger max latest comments
    blogger_blog.comments.latest_comment_limit: 10
```

Il faut enfin afficher les derniers commentaires dans le template de la barre latérale. Mettez à jour le template dans `src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig` en y ajoutant ce qui suit.

```
{# src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig #}

{# .. #}

<section class="section">
    <header>
        <h3>Latest Comments</h3>
    </header>
    {% for comment in latestComments %}
        <article class="comment">
            <header>
                <p class="small"><span class="highlight">{{ comment.user }}</span> commented on
                    <a href="{{ path('BloggerBlogBundle_blog_show', { 'id': comment.blog.id }) }}#comment-{{ comment.id }}">
                        {{ comment.blog.title }}
                    </a>
                    [<em><time datetime="{{ comment.created|date('c') }}">{{ comment.created|date('Y-m-d h:iA') }}</time></em>]
                </p>
            </header>
            <p>{{ comment.comment }}</p>
            </p>
        </article>
    {% else %}
        <p>There are no recent comments</p>
    {% endfor %}
</section>
```

Si vous mettez maintenant à jour le site, vous verrez que les derniers commentaires sont affichés dans la barre latérale, juste en dessous du nuage de tags.

Barre latérale - Nuage de tags et derniers commentaires.

Extensions Twig

Pour le moment nous avons affiché les dates dans un format de date standard tel que *2011-04-21*. Une approche bien plus sympa serait d'afficher depuis combien de temps les commentaires ont été ajoutés, tel que *posté il y a 3 heures*. Nous pourrions ajouter une méthode dans l'entité `Comment` afin de réaliser cela et changer les templates pour utiliser cette méthode au lieu de `{{ comment.created|date('Y-m-d h:iA') }}`.

Comme il est possible que l'on veuille utiliser cette fonctionnalité à d'autres endroits, il est logique de sortir le code de l'entité `Comment`. Comme transformer la date est une tâche spécifique à la vue, nous devrions l'implémenter en utilisant le moteur de template Twig. Twig nous permet en effet cela grâce à ses possibilités d'extensions.

Nous pouvons utiliser l'interface d'extension de Twig pour étendre les fonctionnalités par défaut qu'il propose. Nous allons créer une extension qui nous fournira un nouveau filtre qui s'utilisera de la manière suivante :

```
{{ comment.created|created_ago }}
```

Cela affichera une date de création du commentaire de type *posted 2 days ago* pour *Posté il y a 2 jours*.

L'extension

Créez un fichier pour l'extension Twig dans `src/Blogger/BlogBundle/Twig/Extensions/BloggerBlogExtension.php` et mettez le à jour avec le contenu suivant.

```
<?php
// src/Blogger/BlogBundle/Twig/Extensions/BloggerBlogExtension.php

namespace Blogger\BlogBundle\Twig\Extensions;

class BloggerBlogExtension extends \Twig_Extension
```

```
{
    public function getFilters()
    {
        return array(
            'created_ago' => new \Twig_Filter_Method($this, 'createdAgo'),
        );
    }

    public function createdAgo(\DateTime $dateTime)
    {
        $delta = time() - $dateTime->getTimestamp();
        if ($delta < 0)
            throw new \Exception("createdAgo is unable to handle dates in the future");

        $duration = "";
        if ($delta < 60)
        {
            // Seconds
            $time = $delta;
            $duration = $time . " second" . (($time > 1) ? "s" : "") . " ago";
        }
        else if ($delta <= 3600)
        {
            // Mins
            $time = floor($delta / 60);
            $duration = $time . " minute" . (($time > 1) ? "s" : "") . " ago";
        }
        else if ($delta <= 86400)
        {
            // Hours
            $time = floor($delta / 3600);
            $duration = $time . " hour" . (($time > 1) ? "s" : "") . " ago";
        }
        else
        {
            // Days
            $time = floor($delta / 86400);
            $duration = $time . " day" . (($time > 1) ? "s" : "") . " ago";
        }

        return $duration;
    }

    public function getName()
    {
        return 'blogger_blog_extension';
    }
}
```

Créer l'extension est assez simple. On surcharge la méthode `getFilters()` pour renvoyer autant de filtres que l'on souhaite. Dans le cas présent, on a créé le filtre `created_ago`. Ce filtre est ensuite enregistré de manière à appeler la méthode `createdAgo`, qui se charge simplement de transformer un objet `DateTime` en une chaine de caractères qui représente la durée écoulée depuis la valeur stockée dans l'objet `DateTime`.

Enregistrer l'extension

Pour rendre l'extension Twig disponible, il faut mettre à jour le fichier de services dans `src/Blogger/BlogBundle/Resources/config/services.yml` avec ce qui suit.

```
services:
    blogger_blog.twig.extension:
        class: Blogger\BlogBundle\Twig\Extensions\BloggerBlogExtension
        tags:
            - { name: twig.extension }
```

Vous pouvez voir que celà enregistre un nouveau service en utilisant la classe d'extension `BloggerBlogExtension` que nous venons de créer.

Mettre à jour la vue

Le nouveau filtre Twig est désormais prêt à être utilisé. Mettons à jour la section des derniers commentaires de la barre latérale pour nous en servir. Mettez à jour le contenu du template de la barre latérale dans `src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig` avec ce qui suit :

```
{# src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig #}
```

```
{# .. #}

<section class="section">
    <header>
        <h3>Latest Comments</h3>
    </header>
    {% for comment in latestComments %}
        {# .. #}
        <em><time datetime="{{ comment.created|date('c') }}">{{ comment.created|created_ago }}</time></em>
        {# .. #}
    {% endfor %}
</section>
```

Si vous vous rendez maintenant sur la page d'accueil `http://symblog.dev/app_dev.php/`, vous allez voir que les dates des derniers commentaires utilisent le filtre Twig pour afficher les durées depuis lesquelles ils ont été postés.

Nous allons également mettre à jour les commentaires de la page d'affichage des articles afin d'utiliser là aussi le nouveau filtre. Remplacez le contenu du template dans `src/Blogger/BlogBundle/Resources/views/Comment/index.html.twig` avec ce qui suit.

```
{# src/Blogger/BlogBundle/Resources/views/Comment/index.html.twig #}

{% for comment in comments %}
    <article class="comment {{ cycle(['odd', 'even'], loop.index0) }}" id="comment-{{ comment.id }}">
        <header>
            <p><span class="highlight">{{ comment.user }}</span> commented <time datetime="{{ comment.created|date('c') }}">{{ comment.crea
        </header>
        <p>{{ comment.comment }}</p>
    </article>
{% else %}
    <p>There are no comments for this post. Be the first to comment...</p>
{% endfor %}
```

> **Tip**
>
> Il y a plusieurs extensions Twig utiles disponibles via la librarie Twig-Extensions sur GitHub. Si vous créez une extension utile, proposez une proposition d'ajout (pull request) dans ce dépôt et il est possible qu'elle soit incluse afin que d'autres puissent s'en servir.

Slugification de l'URL

Actuellement, l'URL de chaque article montre seulement l'identifiant de l'article. Bien que ce soit parfaitement acceptable d'un point de vue fonctionnel, c'est pas terrible d'un point de vue SEO (Search Engine Optimization: optimisation pour les moteurs de recherche). Par exemple, l'URL `http://symblog.dev/1` ne donne aucune information sur le contenu de l'article, alors que quelquechose comme `http://symblog.dev/1/a-day-with-symfony2` est beaucoup mieux de ce point de vue. Pour réaliser celà, il nous faut slugifier le titre des articles et nous en servir comme élément de l'adresse. Slugifier le titre revient à enlever tous les caractères non ASCII et les remplacer par un `-`.

Mise à jour de la route

Pour commencer, modifions les règles de routage pour la page d'affichage des articles afin d'ajouter sa nouvelle composante `slug`. Mettez à jour les règles de rouatge dans `src/Blogger/BlogBundle/Resources/config/routing.yml`

```
# src/Blogger/BlogBundle/Resources/config/routing.yml

BloggerBlogBundle_blog_show:
    pattern:  /{id}/{slug}
    defaults: { _controller: BloggerBlogBundle:Blog:show }
    requirements:
        _method:  GET
        id: \d+
```

Le controlleur

Comme avec le composant déjà existant `id`, le nouvel élément `slug` va être passé à l'action du controlleur en argument. Il faut donc mettre à jour le controlleur dans `src/Blogger/BlogBundle/Controller/BlogController.php` afin de répercuter ce changement.

```
// src/Blogger/BlogBundle/Controller/BlogController.php

public function showAction($id, $slug)
```

```
{
    // ..
}
```

Tip

L'ordre dans lequel les arguments sont passés à l'action du controlleur n'a pas d'importance, seul leur nom compte. Symfony2 est capable d'associer les paramètres de routage avec la liste de paramètres pour nous. Bien que nous n'ayons pas utilisé pour le moment de valeurs par défaut, cela vaut le coup de les mentionner ici. Si nous ajoutions un nouveau composant à la règle de routage, nous pourrions très bien lui spécifier également une valeur par défaut, à l'aide de l'option `defauts`.

```
BloggerBlogBundle_blog_show:
    pattern:  /{id}/{slug}/{comments}
    defaults: { _controller: BloggerBlogBundle:Blog:show, comments: true }
    requirements:
        _method:  GET
        id: \d+
```

```
public function showAction($id, $slug, $comments)
{
    // ..
}
```

En utilisant cette méthode, les requêtes à l'adresse `http://symblog.dev/1/symfony2-blog` mèneraient à avoir `$comments` à true dans `showAction`.

Slugification du titre

Comme on veut générer le slug à partir du titre de l'article, nous allons générer automatiquement cette valeur. Nous pourrions réaliser celà automatiquement à l'exécution sur le titre de l'article, mais à la place nous allons plutôt stocker le slug dans l'entité `Blog` et le stocker dans la base de données.

Mise à jour de l'entité Blog

Ajoutons un nouveau membre à l'entité `Blog` pour stocker le slug. Mettez à jour l'entité `Blog` dans `src/Blogger/BlogBundle/Entity/Blog.php`

```
// src/Blogger/BlogBundle/Entity/Blog.php

class Blog
{
    // ..

    /**
     * @ORM\Column(type="string")
     */
    protected $slug;

    // ..
}
```

Générez maintenant les accesseurs pour le nouveau membre `$slug`. Comme avant, lancez la tâche :

```
$ php app/console doctrine:generate:entities Blogger
```

Il est ensuite temps de mettre à jour le schéma de base de donnée :

```
$ php app/console doctrine:migrations:diff
$ php app/console doctrine:migrations:migrate
```

Pour générer la valeur du slug, nous allons utiliser la méthode slugify du Tutorial Symfony 1 Jobeet. Ajoutez la méthode `slugify` dans l'entité `Blog` situé dans `src/Blogger/BlogBundle/Entity/Blog.php`

```
// src/Blogger/BlogBundle/Entity/Blog.php

public function slugify($text)
{
    // replace non letter or digits by -
    $text = preg_replace('#[^\\pL\d]+#u', '-', $text);
```

```
    // trim
    $text = trim($text, '-');

    // transliterate
    if (function_exists('iconv'))
    {
        $text = iconv('utf-8', 'us-ascii//TRANSLIT', $text);
    }

    // lowercase
    $text = strtolower($text);

    // remove unwanted characters
    $text = preg_replace('#[^-\w]+#', '', $text);

    if (empty($text))
    {
        return 'n-a';
    }

    return $text;
}
```

Comme nous voulons générer automatiquement le slug à partir du titre, on peut générer le slug lorsque la valeur du titre est affectée. Pour celà, on peut mettre à jour l'accesseur `setTitle` pour mettre également à jour la valeur du slug. Mettez à jour l'entité `Blog` dans `src/Blogger/BlogBundle/Entity/Blog.php` avec ce qui suit.

```
// src/Blogger/BlogBundle/Entity/Blog.php

public function setTitle($title)
{
    $this->title = $title;

    $this->setSlug($this->title);
}
```

Maintenant mettez à jour la méthode `setSlug` afin d'affecter une valeur *slugifiée* à l'attribut slug.

```
// src/Blogger/BlogBundle/Entity/Blog.php

public function setSlug($slug)
{
    $this->slug = $this->slugify($slug);
}
```

Maintenant rechargez les données factices pour générer les slugs des articles.

```
$ php app/console doctrine:fixtures:load
```

Mise à jour des routes générées

Il faut enfin mettre à jour les appels déjà existants à la génération de route vers la page d'affichage des articles. Il y a plusieurs endroits où celà doit être mis à jour.

Ouvrez le template de la page d'accueil dans `src/Blogger/BlogBundle/Resources/views/Page/index.html.twig` et remplacez son contenu avec ce qui suit. Il y a 3 modifications de la route `BloggerBlogBundle_blog_show` dans ce template. Les modifications ajoutent simplement le slug des titres des articles en paramètre de la fonction `path`.

```
{# src/Blogger/BlogBundle/Resources/views/Page/index.html.twig #}

{% extends 'BloggerBlogBundle::layout.html.twig' %}

{% block body %}
    {% for blog in blogs %}
        <article class="blog">
            <div class="date"><time datetime="{{ blog.created|date('c') }}">{{ blog.created|date('l, F j, Y') }}</time></div>
            <header>
                <h2><a href="{{ path('BloggerBlogBundle_blog_show', { 'id': blog.id, 'slug': blog.slug }) }}">{{ blog.title }}</a></h2>
            </header>

            <img src="{{ asset(['images/', blog.image]|join) }}" />
            <div class="snippet">
```

```
                <p>{{ blog.blog(500) }}</p>
                <p class="continue"><a href="{{ path('BloggerBlogBundle_blog_show', { 'id': blog.id, 'slug': blog.slug }) }}">Continue read
            </div>

            <footer class="meta">
                <p>Comments: <a href="{{ path('BloggerBlogBundle_blog_show', { 'id': blog.id, 'slug': blog.slug }) }}#comments">{{ blog.com
                <p>Posted by <span class="highlight">{{ blog.author }}</span> at {{ blog.created|date('h:iA') }}</p>
                <p>Tags: <span class="highlight">{{ blog.tags }}</span></p>
            </footer>
        </article>
    {% else %}
        <p>There are no blog entries for symblog</p>
    {% endfor %}
{% endblock %}
```

De plus, une mise à jour doit être faite à la section Derniers commentaires de la barre latérale dans le template `src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig`.

```
{# src/Blogger/BlogBundle/Resources/views/Page/sidebar.html.twig #}

{# .. #}

<a href="{{ path('BloggerBlogBundle_blog_show', { 'id': comment.blog.id, 'slug': comment.blog.slug }) }}#comment-{{ comment.id }}">
    {{ comment.blog.title }}
</a>

{# .. #}
```

Enfin, l'action `createAction` du `CommentController` doit être mise à jour lorsqu'elle redirige vers la page d'affichage d'un article lorsqu'un commentaire a été posté. Mettez à jour le `CommentController` situé dans `src/Blogger/BlogBundle/Controller/CommentController.php` avec ce qui suit.

```
// src/Blogger/BlogBundle/Controller/CommentController.php

public function createAction($blog_id)
{
    // ..

    if ($form->isValid()) {
        // ..

        return $this->redirect($this->generateUrl('BloggerBlogBundle_blog_show', array(
            'id'    => $comment->getBlog()->getId(),
            'slug'  => $comment->getBlog()->getSlug())) .
            '#comment-' . $comment->getId()
        );
    }

    // ..
}
```

Maintenant si vous allez sur la page d'accueil `http://symblog.dev/app_dev.php/` et cliquez sur un des titres des articles, vous verrez que le slug des titres des articles est maintenant présent à la fin de l'URL.

Environnements

Les environnements sont à la fois une fonctionnalité très simple et très puissante de Symfony2. Vous n'en êtes peut être pas conscient, mais vous vous en servez depuis le tout premier chapitre de ce tutoriel. Avec les environnements, on peut configurer différents aspects de Symfony2 et de l'application pour qu'elle tourne différemment selon des besoins spécifiques au cours du cycle de vie de l'application. Par défaut, Symfony2 est configuré avec 3 environnement :

1. `dev` - Developpement
2. `test` - Test
3. `prod` - Production

Le rôle de ces environnements est inclus dans leur nom. Lorsque l'on développe une application, il est utile d'avoir la barre de débug à l'acran afin d'avoir des erreurs et des exceptions détaillées, alors qu'en production on ne veut rien de tout cela. En fait, afficher ces informations serait même une faille de sécurité car de nombreux détails relatifs au comportement interne de l'application et du serveur seraient disponibles. En production, il serait plus judicieux d'afficher des pages d'erreur personnalisées avec des messages simples, tout en stockant discrètement les messages d'erreurs dans un fichier log. Il peut également être utile d'activer le cache afin que l'application tourne au maximum de ses capacités. En débug, l'activer serait un véritable cauchemar car il faudrait vider le cache à chaque modification ou presque, ce qui fait au final perdre plus de temps qu'il n'en fait gagner et peut être source d'erreurs.

Le dernier environnement, c'est l'environnement de test. Il est utilisé pour effectuer des tests sur l'application, tel que des tests unitaires ou fonctionnels. Nous n'avons pas parlé des tests pour le moment, mais ils seront abordés en détails dans le chapitre suivant.

Controlleur de facade

Pour le moment dans ce tutoriel, nous avons uniquement utilisé l'environnement de `développement`, ce qui nous avons précisé en utilisant le controlleur de facade `app_dev.php` lorsque nous avons fait des requêtes vers symblog, par exemple `http://symblog.dev/app_dev.php/about`. Si vous regardez le contenu du controlleur de facade de l'environnement de développement dans `web/app_dev.php`, vous y verrez la ligne suivante :

```
$kernel = new AppKernel('dev', true);
```

Cette ligne est celle qui fait démarrer Symfony2. Elle crée une nouvelle instance de l'`AppKernel` de Symfony2, et opte pour l'environnement `dev`.

En comparaison, si vous regardez le controlleur de façade de l'environnement de `production` dans `web/app.php`, vous y verrez :

```
$kernel = new AppKernel('prod', false);
```

Vous pouvez voir que l'environnement `prod` est fourni en paramètre à l'`AppKernel` dans cette instance.

L'environnement de test n'a pas de controlleur de façade, car il n'est pas censé être utilisé dans un navigateur. C'est pourquoi il n'y a pas de fichier `app_test.php`.

Paramètres de configuration

Nous avons vu plus haut comment les controlleurs de façade sont utilisés pour changer l'environnement dans lequel l'application tourne. Nous allons maintenant regarder comment les différents paramètres sont modifiés lorsque l'on utilise tel ou tel environnement. Si vous regardez les fichiers dans `app/config`, vous y verrez plusieurs fichiers `config.yml`. Plus précisémment, il y a un fichier de configuration principal, `config.yml`, et 3 autres qui sont suffixés du nom de l'environnement; `config_dev.yml`, `config_test.yml` et `config_prod.yml`. Chacun de ces fichiers est chargé selon l'environnement courant. Si nous ouvrons le fichier `config_dev.yml`, nous y verrons les lignes suivantes en entête :

```
imports:
    - { resource: config.yml }
```

La directive `imports` va permettre d'importer le contenu du fichier `config.yml` à l'intérieur de celui là. La même directive `import` peut être trouvée au début des 2 autres fichiers de configuration `config_test.yml` et `config_prod.yml`. L'inclusion d'un ensemble commun de paramètres de configuration définis dans `config.yml` permet d'avoir des valeurs spécifiques pour ces paramètres selon les environnements. On peut voir dans le fichier de configuration de l'environnement de développement `app/config/config_dev.yml` les lignes suivantes, qui configurent l'utilisation de la barre de débug :

```
# app/config/config_dev.yml

web_profiler:
    toolbar: true
```

Ce paramètre est absent dans le fichier de configuration de l'environnement de production car nous ne voulons pas que la barre d'outils soit affichée.

Fonctionner en production

Nous allons maintenant voir notre site tourner dans l'environnement de production. Pour cela, il faut tout d'abord vider le cache, à l'aide d'une commande Symfony2 :

```
$ php app/console cache:clear --env=prod
```

Maintenant rendez vous à l'adresse `http://symblog.dev/`. Remarquez qu'il manque le controlleur de façade `app_dev.php`.

> **Note**
>
> Pour ceux qui utilisent les hotes dynamiques virtuels comme dans le lien de la partie 1, il faudra ajouter ce qui suit dans le fichier .htaccess dans `web/.htaccess`.

```
<IfModule mod_rewrite.c>
    RewriteBase /
    # ..
</IfModule>
```

Vous allez remarquer que le site est presque identique, mais un certain nombre d'éléments sont différents. La barre de débug a disparue et les messages d'erreur détaillés ne sont plus affichés : essayez de vous rendre à l'adresse `http://symblog.dev/999` pour vous en assurer.

Production - Erreur 404

Les messages d'exceptions détaillés ont été remplaces par un message plus simple, qui informe l'utilisateur qu'un problème a eu lieu. Ces écrans d'exceptions peuvent être configurés pour s'accorder avec le thème visuel de votre application. Nous reviendrons sur ce sujet dans un futur chapitre.

Vous pouvez également remarquer que le fichier `app/logs/prod.log` se remplit avec des informations sur l'exécution de l'application. C'est un aspect intéressant lorsque vous aurez des problèmes en production mais qu'il n'y aura plus les erreurs et exceptions de l'environnement de développement.

Tip

Comment la requête depuis `http://symblog.dev/` a réussi à emmener jusqu'au fichier `app.php`? Je suis sûr que vous avez tous déjà créé des fichiers tels que `index.html` et `index.php` comme index de sites, mais `app.php` est moins courant; c'est grâce à une des règles du fichier `web/.htaccess` :

```
RewriteRule ^(.*)$ app.php [QSA,L]
```

On peut voir que cette line contient une expression régulière qui associe n'importe quel texte via `^(.*)$` et le fournit à `app.php`.

Vous êtes peut être sur un serveur Apache qui ne dispose pas de `mod_rewrite.c` activé. Dans ce cas, vous pouvez simplement ajouter `app.php` à l'URL, tel que `http://symblog.dev/app.php/`.

Bien que nous ayions couvert les bases de l'environnement de production, nous n'avons pas parlé de plusieurs éléments liés à l'environnement de production, tel que la personnalisation des pages d'erreurs et le déploiement vers un serveur de production à l'aide d'outils tel que capifony. Nous reviendrons plus tard sur ces sujets dans un chapitre ultérieur.

Création de nouveaux environnements

Il est enfin intéressant de savoir que vous pouvez créer vos propres environnements facilement dans Symfony2. Par exemple, vous pouvez avoir envie d'avoir un environnement qui tourne sur le serveur de production mais affiche certaines informations de débug tel que les exceptions. Cela permettrait à la plateforme d'être testée manuellement sur le serveur de production, car les configurations des serveurs de développement et de production peuvent (et c'est souvent le cas) être différentes.

Bien que la création d'un nouvel environnement soit une tâche simple, elle va au delà du cadre de ce tutoriel. Il y a un excellent article article dans le livre de recettes de Symfony2 qui couvre ce sujet.

Assetic

La distribution standard de Symfony2 est accompagnée d'une librairie de gestion des fichiers externes (les assets) appelée Assetic. Cette librairie a été développée par Kris Wallsmith et a été inspirée par la librairie Python webassets.

Assetic se charge de 2 aspects de la gestion des fichiers externes, les assets tels que images, feuilles de style ou fichiers JavaScript, et les filtres qui peuvent être appliqués sur ces assets. Ces filtre permettent de réaliser des tâches utiles tel que la minification des fichiers CSS ou JavaScript, ou bien passer les fichiers CoffeeScript à travers un compilateur, et comibiner les assets ensemble afin de réduire le nombre de requêtes HTTP faites vers le serveur.

Nous avons jusqu'à présent utilisé la fonction Twig `asset` afin d'inclure les fichiers externes, de la manière suivante :

```
<link href="{{ asset('bundles/bloggerblog/css/blog.css') }}" type="text/css" rel="stylesheet" />
```

Ces appels à la fonction `asset` vont être remplacés par Assetic.

Assets

La librairie Assetic décrit un asset de la manière suivante :

Un asset Assetic est quelquechose avec un contenu filtrable qui peut être chargé et déchargé. Cela inclus également les métadonnées, certaines pouvant être manipulées et certaines étant fixées.

Plus simplement, les assets sont des ressources que l'application utilise tel que les feuilles de style et les images.

Feuilles de style

Commençons par remplacer les appels actuels à la fonction `asset` pour les feuilles de styles dans le template principal du `BloggerBlogBundle`. Mettez à jour le contenu du template situé dans `src/Blogger/BlogBundle/Resources/views/layout.html.twig` avec ce qui suit :

```
{# src/Blogger/BlogBundle/Resources/views/layout.html.twig #}

{# .. #}

{% block stylesheets %}
    {{ parent () }}

    {% stylesheets
        '@BloggerBlogBundle/Resources/public/css/*'
    %}
        <link href="{{ asset_url }}" rel="stylesheet" media="screen" />
    {% endstylesheets %}
{% endblock %}

{# .. #}
```

Nous avons remplacé les 2 précédents liens vers les fichiers CSS avec des fonctionnalités Assetic. En utilisant `stylesheets` depuis Assetic, nous avons précisé que toutes les feuilles de style dans `src/Blogger/BlogBundle/Resources/public/css` doivent être combinées en un fichier avant d'être incluses. Combiner plusieurs fichiers est une manière simple mais efficace d'opitmiser le nombre de fichiers nécessaires à votre site web. Moins de fichiers signifie moins de requêtes HTTP vers le serveur. Bien que nous ayons utilisé `*` pour préciser tous les fichiers du répertoire `css`, nous aurions également pu lister chaque fichier individuellement :

```
{# src/Blogger/BlogBundle/Resources/views/layout.html.twig #}

{# .. #}

{% block stylesheets %}
    {{ parent () }}

    {% stylesheets
        '@BloggerBlogBundle/Resources/public/css/blog.css'
        '@BloggerBlogBundle/Resources/public/css/sidebar.css'
    %}
        <link href="{{ asset_url }}" rel="stylesheet" media="screen" />
    {% endstylesheets %}
{% endblock %}

{# .. #}
```

Le résultat final dans les 2 cas est le même. La première option qui utilise `*` assure que les nouveaux fichiers CSS ajoutés dans le répertoire seront ajoutés et combinés dans le fichier CSS d'Assetic. Cela n'est toutefois pas forcément le comportement que l'on souhaite avoir, donc utilisez l'une ou l'autre des méthodes selon vos besoins.

Si vous regardez la sortie HTML via `http://symblog.dev/app_dev.php/`, vous verrez que les fichiers CSS ont été inclus de la manière suivante (remarquez que nous sommes retourné dans l'environnement de développement).

```
<link href="/app_dev.php/css/d8f44a4_part_1_blog_1.css" rel="stylesheet" media="screen" />
<link href="/app_dev.php/css/d8f44a4_part_1_sidebar_2.css" rel="stylesheet" media="screen" />
```

Au premier abord, vous vous demandez peut être quels sont ces 2 fichiers, car nous avons dit plus haut qu'Assetic combinerait les fichiers en 1 fichier. C'est parce que nous sommes dans l'environnement de `developpement`. On peut demander à Assetic de fonctionner en mode non débug We can ask Assetic to run in non-debug mode en mettant le paramètre `debug` à false de la manière suivante :

```
{# src/Blogger/BlogBundle/Resources/views/layout.html.twig #}

{# .. #}

{% stylesheets
    '@BloggerBlogBundle/Resources/public/css/*'
    debug=false
%}
    <link href="{{ asset_url }}" rel="stylesheet" media="screen" />
```

```
{% endstylesheets %}

{# .. #}
```

Si vous regardez maintenant le HTML, vous y verrez ceci :

```
<link href="/app_dev.php/css/3c7da45.css" rel="stylesheet" media="screen" />
```

Si vous regardez le contenu de ce fichier, vous verrez que les 2 fichiers CSS `blog.css` et `sidebar.css` ont été combinés en 1 fichier. Le nom de fichier utilisé pour le fichier généré est produit aléatoirement par Assetic. Si vous voulez controller le nom du fichier généré, utilisez l'option `output` comme suit :

```
{% stylesheets
    '@BloggerBlogBundle/Resources/public/css/*'
    output='css/blogger.css'
%}
    <link href="{{ asset_url }}" rel="stylesheet" media="screen" />
{% endstylesheets %}
```

Avant de continuer, supprimez le paramètre `debug` de l'exemple précédent, car nous voulons revenir au comportement par défaut sur les assets.

Nous devons également mettre à jour le template de base de l'application, dans `app/Resources/views/base.html.twig`.

```
{# app/Resources/views/base.html.twig #}

{# .. #}

{% block stylesheets %}
    <link href='http://fonts.googleapis.com/css?family=Irish+Grover' rel='stylesheet' type='text/css'>
    <link href='http://fonts.googleapis.com/css?family=La+Belle+Aurore' rel='stylesheet' type='text/css'>
    {% stylesheets
        'css/*'
    %}
        <link href="{{ asset_url }}" rel="stylesheet" media="screen" />
    {% endstylesheets %}
{% endblock %}

{# .. #}
```

JavaScripts

Bien que nous n'ayions pas actuellement de fichiers JavaScript dans notre application, leur utilisation via Assetic est très semblable à celle des feuilles de style :

```
{% javascripts
    '@BloggerBlogBundle/Resources/public/js/*'
%}
    <script type="text/javascript" src="{{ asset_url }}"></script>
{% endjavascripts %}
```

Filtres

La vrai puissance d'Assetic vient de ses filtres. Les filtres peuvent être appliqués à des assets ou à un ensemble d'assets. Il y a un grand nombre de filtres à l'intérieur de la librairie de base, qui réalisent les taches courantes suivantes :

1. `CssMinFilter`: minifaction de la CSS
2. `JpegoptimFilter`: optimisation des fichiers JPEGs
3. `Yui\CssCompressorFilter`: compression de fichiers CSS à l'aide de l'outil YUI compressor
4. `Yui\JsCompressorFilter`: compression de fichiers JavaScript à l'aide de l'outil YUI compressor
5. `CoffeeScriptFilter`: compile CoffeeScript en JavaScript

Une liste complète des filtres disponible se trouve dans le Readme Assetic.

Plusieurs de ces filtres passent en fait la main à un autre programme ou à une autre librairie, tel que YUI Compressor, donc il est possible que vous ayiez à installer ou configurer les librairies nécessaires pour utiliser certains filtres.

Téléchargez YUI Compressor, décompressez l'archive et copiez les fichiers du répertoire `build` dans `app/Resources/java/yuicompressor-2.4.6.jar`. Cela suppose que vous ayiez téléchargé la version `2.4.6`, sinon changez le numéro de version en conséquences.

Nous allons ensuite configurer un filtre Assetic pour minifier la CSS à l'aide de YUI Compressor. Mettez à jour la configuration de l'application dans `app/config/config.yml` avec le contenu suivant :

```
# app/config/config.yml

# ..

assetic:
    filters:
        yui_css:
            jar: %kernel.root_dir%/Resources/java/yuicompressor-2.4.6.jar

# ..
```

Nous venons de configurer un filtre `yui_css` qui va utiliser l'exécutable Java de l'outil YUI Compressor, que nous allons placer dans le répertoire des ressources de l'application. Afin d'utiliser ce filtre, il faut lui préciser avec quels assets s'en servir. Mettez à jour le template dans `src/Blogger/BlogBundle/Resources/views/layout.html.twig` pour utiliser le filtre `yui_css`.

```
{# src/Blogger/BlogBundle/Resources/views/layout.html.twig #}

{# .. #}

{% stylesheets
    '@BloggerBlogBundle/Resources/public/css/*'
    output='css/blogger.css'
    filter='yui_css'
%}
    <link href="{{ asset_url }}" rel="stylesheet" media="screen" />
{% endstylesheets %}

{# .. #}
```

Si vous rafraichissez la page d'accueil du site Symblog et regardez les fichiers générés par Assetic, vous verrez qu'ils ont été minifiés. Bien que la minification soit une bonne idée sur un serveur de production, elle peut rendre le débuggage difficile, en particulier lorsque le Javascript est minifié. On peut la désactiver pour l'environnement `development` en préfixant le filtre avec un `?` de la manière suivante.

```
{% stylesheets
    '@BloggerBlogBundle/Resources/public/css/*'
    output='css/blogger.css'
    filter='?yui_css'
%}
    <link href="{{ asset_url }}" rel="stylesheet" media="screen" />
{% endstylesheets %}
```

Génération des assets pour la production

En production, on peut générer les fichiers d'assets grâce à Assetic afin qu'ils deviennent de vrais fichiers prêts à être utilisés sur le serveur web. Le processus de création des assets avec Assetic pour chaque adresse peut être assez long, en particulier si des filtres sont appliqués aux assets. Le sauvegarder de manière définitive pour la production assure qu'Assetic ne sera pas utilisé pour manipuler les assets, mais seulement pour fournir les assets pré-traités. Lancez la commande suivante pour conserver les fichiers assets traités sur le disque :

```
$ app/console --env=prod assetic:dump
```

Vous pouvez remarquer que plusieurs fichiers CSS ont été générés dans le répertoire `web/css`. Si vous lancez Symblog dans l'environnement de production, vous verrez que les fichiers proviennent directement de ce répertoire.

> **Note**
>
> Si vous stockez les fichiers assets sur le disque mais souhaitez retourner dans l'environnement de développement, vous devrez supprimer les fichiers créé dans le répertoire `web/` pour permettre à Assetic de les recréer.

Lecture additionnelle

Nous avons seulement abordé une fraction des possibilités offertes par Assetic. Il y a plus de ressources en ligne, en particulier dans le livre de recettes de Symfony2, en particulier (mais en anglais) :

How to Use Assetic for Asset Management

How to Minify JavaScripts and Stylesheets with YUI Compressor

How to Use Assetic For Image Optimization with Twig Functions

How to Apply an Assetic Filter to a Specific File Extension

Il y a également plusieurs bons articles de Richard Miller tel que :

Symfony2: Using CoffeeScript with Assetic

Symfony2: A Few Assetic Notes

Symfony2: Assetic Twig Functions

> **Tip**
>
> Il est à noter également que Richard Miller a également de nombreux articles très intéressant dans de nombreux dommaines de Symfony2, tel que l'injection de dépendances, les services ainsi que les déjà mentionnés guides sur Assetic. Cherchez les articles taggés avec symfony2

Conclusion

Nous avons couvert plusieurs nouveaux dommaines de Symfony2, tel que les environnements et comment utiliser la librairie Assetic. Nous avons également amélioré la page d'accueil, et ajouté plusieurs composants à la barre latérale.

Dans le prochain chapitre, nous aller passer aux tests. Nous parlerons à la fois des tests unitaires et des tests fonctionnels avec PHPUnit. Nous verrons comment Symfony2 aide grandement à l'écriture des tests avec plusieurs classes pour faciliter l'écriture des tests fonctionnels qui simulent des requêtes, permettre de remplir les formulaires, cliquent sur les liens et nous permettent d'inspecter les réponses obtenues.