Building an app with

Symfony 3

Part3 The template

Création et utilisation des Templates

Templates:

Un template est un simple fichier texte qui va générer tout format de fichiers textes (HTML,XML,CSV,LaTeX …).
Le template le plus courant est un template PhP: un fichier texte généré par PhP qui contient un mix de texte et de code PhP:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Welcome to Symfony!</title>
  </head>
  <body>
    <h1><?php echo $page_title ?></h1>

    <ul id="navigation">
      <?php foreach ($navigation as $item): ?>
        <li>
          <a href="<?php echo $item->getHref() ?>">
            <?php echo $item->getCaption() ?>
          </a>
        </li>
      <?php endforeach ?>
    </ul>
  </body>
</html>
```

Mais Symfony possède dans ses packages un langage puissant de templating appelé Twig.

Twig permet d'écrire des templates concis et lisibles pour agréable pour les web designer et plus puissant que les templates PhP:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Welcome to Symfony!</title>
    </head>
    <body>
        <h1>{{ page_title }}</h1>

        <ul id="navigation">
            {% for item in navigation %}
                <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
            {% endfor %}
        </ul>
    </body>
</html>
```

{{... }} "dit quelque chose": affiche une variable ou le résultat d'un calcul au template
{%...%} "fait quelque chose": une balise qui contrôle la logique du template comme les boucles.
{#...#} "commentaires": équivalent du /* */ en PhP.
Twig contient aussi des filtres qui modifient le contenu avant d'être rendu. {{title|upper}}
D'autres balises et filtres existent, nous reviendrons dessus. Nous pourrons créer nos propres éléments.

Twig aussi supporte les fonctions et de nouvelles peuvent être facilement ajoutées.

Par exemple, dans le script suivant nous utilisons un for standard et la fonction cycle pour afficher 10 balises div, avec une alternance de classes odd et even.

```
{% for i in 0..10 %}
    <div class="{{ cycle(['odd', 'even'], i) }}">
      <!-- some HTML here -->
    </div>
{% endfor %}
```

Pourquoi Twig plutôt que PhP:
Twig est plus simple à manipuler que PhP.
C'est créé pour du design, son système fonctionne avec des présentations expresses, pas de la logique de dev.
Twig permet de réaliser des choses impossibles en PhP comme le contrôle des espaces.
Exemple de combinaison de boucle et de conditionnel:

```
<ul>
   {% for user in users if user.active %}
      <li>{{ user.username }}</li>
   {% else %}
      <li>No users found</li>
   {% endfor %}
</ul>
```

Cache Template Twig

Twig est rapide. Chaque template Twig est compilé en une classe PhP qui est rendu à l'exécution.
Les classes compilées sont localisées dans le répertoire var/cache/{environnement}/twig où {environnement} est l'environnement, soit de dev ou prod. Ce qui permet des fois de débugger notre travail.
En mode dev, un template Twig est automatiquement recompilé.
En mode prod, nous devons vider le cache pour que le template Twig soit recompilé.

Héritage des templates et les layouts

Les templates dans un projet partage des éléments communs, comme le Header, le Footer, la SideBar ou plus.

Dans symfony, le problème est pensé différemment: un Template peut être "décoré" par un autre. Comme pour l'héritage des classes en PhP: l'héritage des templates permet de construire un template "layout" de base qui contient tous les éléments communs au site définit comme blocks.

```
{# app/Resources/views/base.html.twig #}
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>{% block title %}Test Application{% endblock %}</title>
  </head>
  <body>
    <div id="sidebar">
      {% block sidebar %}
        <ul>
          <li><a href="/">Home</a></li>
          <li><a href="/blog">Blog</a></li>
        </ul>
      {% endblock %}
    </div>
    <div id="content">
      {% block body %}{% endblock %}
    </div>
  </body>
</html>
```

Ce Template définit le squelette de base HTML d'une page simple à 2 colonnes.
Dans cet exemple, nous avons 3 zones {% block %} définies (title, sidebar et body). Chaque bloque pour être surchargé par un template enfant ou laissé avec son implémentation.
Ce template peut être affiché comme tel. Dans ce cas title, sidebar et body devraient reprendre les valeurs par défaut.
Un template enfant ressemble à:

```
{# app/Resources/views/blog/index.html.twig #}
{% extends 'base.html.twig' %}
{% block title %}My cool blog posts{% endblock %}
{% block body %}
        {% for entry in blog_entries %}
                <h2>{{ entry.title }}</h2>
                <p>{{ entry.body }}</p>
        {% endfor %}
{% endblock %}
```

Le template parent est identifié par une chaine de caractère spéciale (base.html.twig). Le dossier pointé par défaut est app/Resources/views du projet. On peut aussi utiliser un nommage logique équivalent ::base.html.twig. Nous reviendrons dessus.

Pour hériter d'un template nous utilisons la balise {% extends %}. Qui indique au moteur de template d'évaluer en premier le template de base, qui définit les layouts et les différents blocs. Le template enfant est alors affiché, avec les blocs title et body du parent qui seront remplacés par ceux de l'enfant. En fonction des valeurs de blog_entries, nous aurons ce résultat:

```
<!DOCTYPE html>
<html>
<head>
        <meta charset="UTF-8">
        <title>My cool blog posts</title>
</head>
<body>
        <div id="sidebar">
                <ul>
                        <li><a href="/">Home</a></li>
                        <li><a href="/blog">Blog</a></li>
                </ul>
        </div>
        <div id="content">
                <h2>My first post</h2>
                <p>The body of the first post.</p>
                <h2>Another post</h2>
                <p>The body of the second post.</p>
        </div>
</body>
</html>
```

On remarque que le template enfant ne définit pas le bloc sidebar, c'est la valeur définit dans le parent qui est utilisé. On utilise toujours une balise {% block %} par défaut dans un template parent.

On peut utiliser autant de niveaux d'héritage que l'on veut. Nous verrons un modèle d'héritage à 3 niveaux pour en expliquer le fonctionnement sous Symfony.

- La balise {% extends %} doit être la première dans un template.
- Plus vous aurez de blocs {% block %} dans votre template de base, plus votre layout sera flexible.
- Si vous avez du contenu dupliqué dans des templates, c'est que vous devez utiliser le bloc {% block %} dans un template parent. Dans certain cas, il est judicieux de créer ce contenu dans un nouveau template que vous inclurez.
- Si vous avez besoin du contenu d'un bloc d'un template parent, vous pouvez utiliser la fonction {{parent()}}. C'est pratique pour ajouter du contenu à un bloc parent sans la réécrire.

```
{% block sidebar %}
        <h3>Table of Contents</h3>
        {# ... #}
        {{ parent() }}
{% endblock %}
```

Localisation et nommage des templates

Par défaut, les templates sont situés dans deux endroits différents:
app/Resources/views/
C'est le répertoire des vues de l'application.
path/to/bundle/Resources/views/
C'est le répertoire de partage de bundle des vues.
Vous utiliserez surtout app/Resources/views. Ce path sera relatif à son répertoire. Par exemple, pour afficher ou hériter de app/Resources/vieuw/base.html.twig, vous devrez utiliser le path base.html.twig et affiché ou hériter app/Resources/views/blog/index.html.twig vous utiliserez le path blog/index.html.twig

Référencer des templates dans un bundle.

Symfony utilise un syntaxe texte bundle:directory:filename pour les templates à l'intérieur d'un bundle.
Cela permet pour plusieurs types de templates, d'avoir chacun une location spécifique:

- AcmeBlogBundle:Blog:index.html.twig: Cette syntaxe est utilisée pour spécifier le template d'une page spécifique. Les trois parties de la chaîne de caractère sont séparées par (:) qui veulent dire:
  - AcmeBlogBundle: (bundle) le template est situé à l'intérieur d'AcmeBlogBundle (src/Acme/BlogBundle).
  - Blog: (répertoire / directory) indique que le template est situé dans le sous répertoire Resources/views.
  - index.html.twig (nom du fichier / filename) le nom actuel du fichier est index.html.twig.

  Sachant que le bundle AcmeBlogBundle est dans src/Acme/BlogBundle, le chemin final pour le layout est Resources/views/layout.html.twig
- AcmeBlogBundle::layout.html.twig: cette syntaxe réfère au template de base spécifique à AcmeBlogBundle. il manque la partie répertoire Blog, le template est à Resources/view/layout.html.twig dans AcmeBlogBundle. Il y a 2 infos au milieu de la chaîne de caractère quand le sous répertoire du contrôleur manque.

Suffixe des Templates:

Chaque nom de template pour avoir 2 extensions qui indiquent le format et le moteur:

Filename	Format	Moteur
Blog/index.html.twig	HTML	Twig
blog/index.html.php	HTML	PhP
blog/index.css.twig	CSS	Twig

Par défaut, les templates symfo peuvent être écrit soit en Twig soit en PhP et la dernière partie de l'extension (Twig ou PhP) spécifient le moteur utilisé. La première partie de l'extension (html ou css ...) est le format final que le template va généré.

Balises et Helpers

Nous allons voir comment utiliser les outils pour nous permettre de réaliser le plus commun des templates tout en incluant d'autres, liant les pages et incluant des images.
Symfony est fournit avec de bundle qui ont des balises Twig spécialisées qui facilite le travaildu template designer. En Php, le système de templating nous fournit un système extensible **helper** qui propose des outils pratiques dans le contexte du template.
En Twig, nos balises sont {% block %}, alors qu'en PhP elles sont de ce format ($view['slots']).

Inclure d'autres templates:

Pour réutiliser un gros morceau de PhP, vous créez habituellement une nouvelle classe ou fonction. On fait de même avec les templates. En déplaçant le template réutilisé dans son propre tempalte, il peut comme ça être inclus dans n'importe quel autre template.
Template qui sera réutilisé:

```
{# app/Resources/views/article/article_details.html.twig #}
<h2>{{ article.title }}</h2>
<h3 class="byline">by {{ article.authorName }}</h3>
<p>
        {{ article.body }}
</p>
```

Incluons ce template dans un autre:

```
{# app/Resources/views/article/list.html.twig #}
{% extends 'layout.html.twig' %}
{% block body %}
        <h1>Recent Articles<h1>
        {% for article in articles %}
                {{ include('article/article_details.html.twig', { 'article': article }) }}
        {% endfor %}
{% endblock %}
```

On inclut le template avec la fonction {{include()}}.
Le template article_details.html.twig utilise la variable article. Toutes les variables dans list.html.twig sont dans article_details.html.twig. {'article':article} est de format Twig. Pour passer plusieurs éléments, vous devez avoir {'foo':foo,'bar':bar}.

Incorporation par contrôleur:

Dans certain cas, vous aurez besoin de faire plus qu'inclure un simple template. Imaginons que vous ayez une sidebar dans votre layout qui contient les trois plus récents articles. Récupérer les 3 articles sous entend une requête dans la bdd et une logique qui ne peut être fait dans un template.
Le principe est de simplement inclure le résultat d'un contrôleur depuis le template.
ex: créons le contrôlmeur qui renvoi un certain nombre d'articles récents:

```
// src/AppBundle/Controller/ArticleController.php
namespace AppBundle\Controller;
// ...
class ArticleController extends Controller
{
            public function recentArticlesAction($max = 3)
            {
                        // make a database call or other logic
                        // to get the "$max" most recent articles
                        $articles = ...;
                        return $this->render(
                                    'article/recent_list.html.twig',
                                    array('articles' => $articles)
                        );
            }
}
```

Le template recent_list est:

```
{# app/Resources/views/article/recent_list.html.twig #}
{% for article in articles %}
            <a href="/article/{{ article.slug }}">
                        {{ article.title }}
            </a>
{% endfor %}
```

L'URL est codé en dure, ce qu'il faut éviter.
Pour inclure le contrôleur, vous aurez besoin de vous y référencer en utilisant une chaîne de caractère standard pour les contrôleurs (bundle:controller:action):

```
{# app/Resources/views/base.html.twig #}
{# ... #}
<div id="sidebar">
            {{ render(controller(
                        'AppBundle:Article:recentArticles',
                        { 'max': 3 }
            )) }}
</div>
```

Liens entre pages:

Créer des liens vers d'autres pages dans votre app est un boulot courant pour un template. Plutôt que de coder en dur vos URLs dans un template, utilisez la fonction **Twig** path (ou le **helper** en PhP) pour générer les URLs basées sur la configuration de routage. Comme ça vous pouvez modifier plus tard l'URL d'un page, juste en changeant la configuration du routage; les templates généreront automatiquement la nouvelle URL.

En premier, lions nous vers la page "_welcome" accessible depuis la configuration de route suivante:

```
// src/AppBundle/Controller/WelcomeController.php
// ...
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
class WelcomeController extends Controller
{
    /**
    * @Route("/", name="_welcome")
    */
    public function indexAction()
    {
        // ...
    }
}
```

Pour lier vers la page, utilisez la fonction Twig **path** et indiquez la route:

```
<a href="{{ path('_welcome') }}">Home</a>
```

Comme prévu, cela va générer l'URL /

Pour une route plus complexe:

```
// src/AppBundle/Controller/ArticleController.php
// ...
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
class ArticleController extends Controller
{
    /**
    * @Route("/article/{slug}", name="article_show")
    */
    public function showAction($slug)
    {
        // ...
    }
}
```

Dans ce cas, nous devons spécifier le nom de la route (article_show) et une valeur pour le paramètre {slug}

```
{# app/Resources/views/article/recent_list.html.twig #}
{% for article in articles %}
    <a href="{{ path('article_show', {'slug': article.slug}) }}">
        {{ article.title }}
    </a>
{% endfor %}
```

Lier avec des assets

Les templates aussi se réfèrent aux images, JS, CSS et autres assets. Vous pouvez les coder en dur (/images/logo.png) mais symfo fournit une fonction Twig plus dynamique **asset**:

```
<img src="{{ asset('images/logo.png') }}" alt="Symfony!" />

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

La principale fonction d'asset est rendre plus portable nos liens. Si l'app est installé dans la racine (http://www.example.com) alors le chemin créé est /images/logo.png. Mais si votre app est dans un sous-dossier http://www.example.com/my_app/ chaque chemin d'asset sera alors /my_app/images/logo.png.

La fonction asset permet aussi d'envoyer une requête texte à votre asset par ex /images/logo?v2.

```
<img src="{{ asset('images/logo.png', version='3.0') }}" alt="Symfony!" />
```

Inclure du CSS et du Js dans Twig

Tout site possède du CSS et du JS. Dans symfo, inclure ces assets est réalisé par l'héritage des templates. Commençons par ajouter deux blocs dans notre template de base qui veut contenir nos assets: un appelè stylesheets à l'intérieur de head et un autre nommé javascripts juste avant la fermeture de la balise body:

```
{# app/Resources/views/base.html.twig #}
<html>
        <head>
                {# ... #}
                {% block stylesheets %}
                        <link href="{{ asset('css/main.css') }}" rel="stylesheet" />
                {% endblock %}
        </head>
        <body>
                {# ... #}
                {% block javascripts %}
                        <script src="{{ asset('js/main.js') }}"></script>
                {% endblock %}
        </body>
</html>
```

Si nous avons besoin d'inclure en plus du css ou du js d'un template enfant.
Imaginons que vous ayez une page de contact et que vous ayez besoin d'inclure contact.css juste sur cette page. Depuis le template de la page contact, faites:

```
{# app/Resources/views/contact/contact.html.twig #}
{% extends 'base.html.twig' %}
{% block stylesheets %}
        {{ parent() }}
        <link href="{{ asset('css/contact.css') }}" rel="stylesheet" />
{% endblock %}
{# ... #}
```

Dans ce template, vous surécrivez le bloc stylesheets et mettez votre nouvelle balise css à l'intérieur de ce bloc.

Héritage à 3 niveaux

Il est d'usage courant d'utiliser l'héritage à 3 niveaux.

Prenons un exemple:

- Créons un fichier app/Resources/views/base.html.twig qui contient le layout principal pour votre app.

- Créons un template pour chaque section de votre site. Par exemple, la fonctionnalité du blog devrait avoir un template appelé blog/layout.html.twig qui contient uniquement les éléments spécifique au blog:

```
{# app/Resources/views/blog/layout.html.twig #}
{% extends 'base.html.twig' %}
{% block body %}
    <h1>Blog Application</h1>
    {% block content %}{% endblock %}
{% endblock %}
```

- Créons des templates individuels pour chaque page et héritons le template de section approprié. Par exemple, la page index devrait être nommé quelque chose de proche de blog/index.html.twig et liste les posts du blog actuel:

```
{# app/Resources/views/blog/index.html.twig #}
{% extends 'blog/layout.html.twig' %}

{% block content %}
    {% for entry in blog_entries %}
        <h2>{{ entry.title }}</h2>
        <p>{{ entry.body }}</p>
    {% endfor %}
{% endblock %}
```

Ce template hérite du template section (layout) qui lui-même hérite du layout de base de l'application (base.html.twig).