Faculté
des Sciences
Département des Sciences Informatiques

UMONS
Université de Mons

Générateur Dynamique de Contenu Statique

Projet de 1^{ere} Master réalisé par Maxime DE WOLF

Année académique 2017–2018

Directeur: Decan Alexandre

Service: Sciences Informatiques

Faculté des Sciences • Université de Mons • Place du Parc 20 • B-7000 Mons

Table des matières

1 Introduction

Dans le cadre du cours de projet et lecture et rédactions scientifiques, il nous est demandé de réaliser un projet d'envergure relativement importante. Dans notre cas, il s'agit d'implémenter un générateur de contenu statique. Ce rapport a pour but de vous présenter l'avancement de ce projet.

Il existe déjà énormément de générateurs de contenu statique disponibles sur Internet. StaticGen [1] établit un classement des générateurs de contenu statique *open-source* suivant leur popularité sur GitHub. Néanmoins, ces générateurs, sont souvent dédiés à une tâche spécifique ce qui limite donc leur utilisation.

L'objectif du générateur de contenu statique que nous développons est de pouvoir être utilisé pour diverses tâches. Il doit donc être paramétrable afin de s'adapter à un grand nombre de situations. Afin d'y parvenir, nous utilisons un système de règles relativement souple pour donner à l'utilisateur un contrôle total sur le comportement du générateur de contenu statique. Nous permettons également à l'utilisateur d'y ajouter de nouvelles fonctionnalités qu'il aura lui-même créées.

Dans un premier temps, nous dresserons le cahier des charges de ce projet. Cela consistera à lister les fonctionnalités nécessaires à respecter afin que ce projet soit mené à bien. Nous illustrerons également quelques cas d'utilisations afin de clarifier le cahier des charges.

Ensuite, nous listerons rapidement quelques générateurs de contenu statique existants. Nous exhiberons leurs différences ainsi que leurs fonctionnalités.

Troisièmement, nous dresserons une liste de l'ensemble des fonctionnalités que devra posséder notre générateur de contenu statique. Nous en profiterons également pour le comparer aux autres générateurs présentés en Section 3. Cela nous permettra donc de justifier la nécessité de ce projet.

Enfin, nous parlerons des techniques mises en place pour répondre aux besoins présentés en Section 4. En d'autres termes, cette section détaillera l'approche qui sera utilisée lors du développement de notre générateur de contenu statique.

2 Cahier des charges

Cette section a pour rôle de présenter le cahier des charges de ce projet. Nous introduisons également le concept de générateur de contenu statique. La seconde partie de cette section liste quelques cas d'utilisation afin d'illustrer le fonctionnement souhaité par ce projet. Notez que les fonctionnalités en elles-même seront détaillées dans la Section 4.

Le cahier des charges de ce projet est volontairement resté assez flou. En effet, les seules consignes données sont de concevoir un générateur de contenu statique qui donne à l'utilisateur une grande souplesse d'utilisation afin de pouvoirs effectuer des tâches diverses. Ce générateur de contenu statique doit aussi être simple à utiliser pour le "grand public" et ne doit donc pas faire appel à des connaissances en programmation. Afin de préciser ce que sont les "tâches diverses" que ce générateur doit être capable d'accomplir, quelques cas d'utilisations sont illustrés dans cette section.

2.1 Générateur de contenu statique

Nous introduisons ici les notations que nous allons utiliser tout au long de ce rapport. Cette sous-section contient également notre définition d'un générateur de contenu statique.

Définition 1. *Un **générateur de contenu** sert à lier des données à un template (ou gabarit) afin de générer du contenu (document, site web, ...). Il sera qualifié de **statique** si ce contenu est identique peu importe le contexte dans lequel il est consulté. Les données, les templates et les contenus sont appelés **paramètres** d'un générateur de contenu statique dans le reste de ce rapport.*

Définition 2. *Un **item** est la brique de base d'un paramètre. Il s'agit d'une abstraction de ce qu'un paramètre représente comme valeur. En pratique un item peut être une page web, un fichier texte, une base de données, une structure de données, ...*

Définition 3. *La **multiplicité** des paramètres d'un générateur de contenu statique est le nombre d'items qu'ils représentent.*

Remarque 1. *Nous utilisons parfois les termes "générateur de contenu statique" et "générateur" de manière interchangeable afin d'éviter les répétitions et les lourdeurs d'écritures.*

A savoir que les générateurs de contenu statique sont très utilisés pour produire des sites web statiques. De ce cas, les *CMS* (ou SGC) peuvent être considérés comme des générateurs de contenu statique. Ils offrent toutefois plus de fonctionnalités qu'un générateur classique mais cela s'écarte du cadre de ce projet. Cela n'est vrai que pour les *CMS* qui ne génère que du contenu statique.

Notre générateur de contenu statique a une vocation un peu plus générale car il permettra de générer à peu près n'importe quel type de contenu statique. Leur utilisation est souvent liée au concept de *templates* qui permettent de séparer les informations de la mise en page en elle-même. La Figure 1 montre une utilisation type de ce genre de générateur. L'Exemple 1 illustre cela grâce à un cas simple et concret.

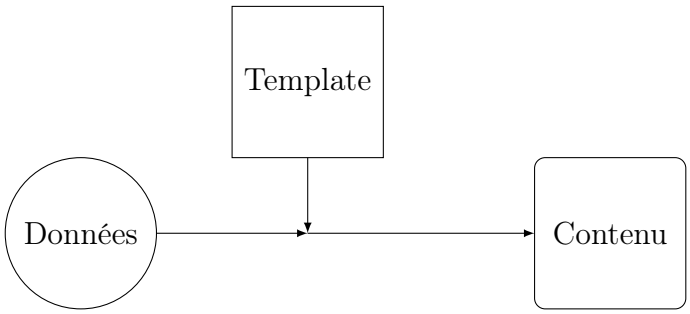

FIGURE 1 – Utilisation type d'un générateur de contenu statique

Exemple 1. *Les Figures 2, 3, 4 représentent respectivement les données à utiliser, le template à appliquer et le contenu ainsi généré. Nous pouvons ainsi constater que le générateur de contenu statique génère ce dernier en remplaçant les champs du template par les valeurs contenues dans le fichier de données.*

2.2 Cas d'utilisations

Nous distinguons principalement trois cas d'utilisations propres aux générateurs de contenu statique. Nous appelons ces trois cas comme suit : *OneToMany*, *ManyToOne* et *ManyToMany*. En pratique, ces trois cas se distinguent par la multiplicités des paramètres du générateur de contenu statique. Chacun d'entre eux est illustré à l'aide d'une figure explicitant leurs multiplicités.

```
{
    "players": [
        {
            "name": "Farar",
            "class": "rôdeur",
            "race": "humain",
            "level": "1"
        },
        {
            "name": "Börin",
            "class": "moine",
            "race": "nain",
            "level": "1"
        }
    ]
}
```

FIGURE 2 – Données au format JSON

```
Liste des joueurs:

{% for player in current.players %}
    -> {{ player.name }}:
        {{player.class}} {{player.race}} de niveau {{ player.level }}

{% endfor %}
```

FIGURE 3 – Template au format Jinja2

```
Liste des joueurs:

    -> Farar:
        rôdeur humain de niveau 1

    -> Börin:
        moine nain de niveau 1
```

FIGURE 4 – Contenu au format *txt*

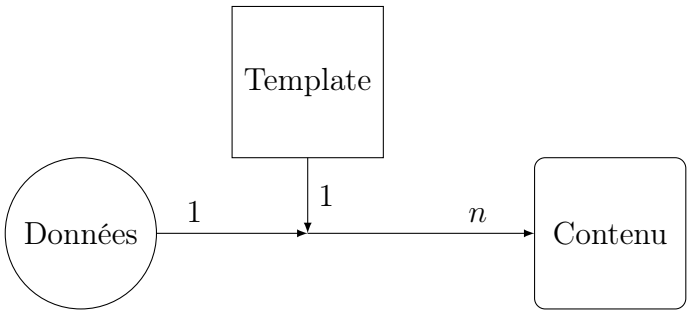

FIGURE 5 – Représentation du cas **OneToMany**.

Nous présentons également 2 cas d'utilisations supplémentaires qui ne sont pas directement réalisables grâce à notre générateur de contenu statique. Nous présentons donc aussi des techniques pour contourner ces limitations. Ces cas d'utilisations sont appelés *ManyTemplatesIn* et *ManyTemplatesOut*.

OneToMany

La cas *OneToMany* désigne l'utilisation d'un générateur de contenu statique en vue de générer plusieurs *items* de contenus à partir d'un seul *item* de données. Ce cas d'utilisation est illustré par la Figure 5 en considérant que **n** est la multiplicité des contenus à générer. L'Exemple 2 met en pratique ce cas d'utilisation.

Exemple 2. *Par exemple, pour un site de vente, cela correspond à générer autant de page web qu'il y a de produits, chaque page ne contenant les caractéristiques que d'un unique produit (items de contenu). L'item de données est ici représenté par une base de données contenant les caractéristiques de tous les produits. Dans cet exemple, le template est appliqué à chaque produit individuellement pour générer une page web par produit.*

ManyToOne

Le second cas, *ManyToOne*, est le cas inverse de *OneToMany*. En effet, ce cas désigne l'utilisation d'un générateur de contenu statique qui génère un seul *item* à partir de plusieurs. Cela donne la Figure 6 où **n** est la multiplicité des données. L'Exemple 3 met ce cas d'utilisation en pratique.

Exemple 3. *Par exemple, pour un blog, cela correspond à générer une page web contenant l'ensemble de tous les posts (item de contenu). Celle-ci serait*

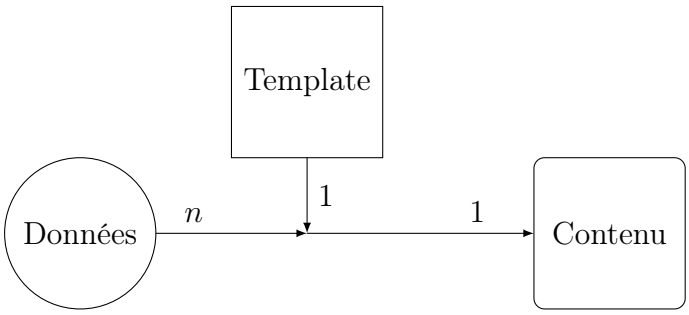

FIGURE 6 – Représentation du cas **ManyToOne**.

générée à partir d'un ensemble de fichier contenant chacun un post (items de données). Dans ce cas, le template est appliqué une fois sur l'ensemble des posts dans le but de générer une unique page web.

ManyToMany

Le dernier cas est celui que nous appellerons *ManyToMany*. Ce cas est en fait une généralisation des deux cas précédents. Effectivement, *ManyToMany* désigne l'utilisation d'un générateur de contenu statique en vue de générer plusieurs *items* de contenu à partir de plusieurs *items* de données. La Figure 7 permet une vision plus claire de ce cas où **n** désigne la multiplicité des données et **m** la multiplicité des contenus. L'Exemple 4 met en pratique ce cas d'utilisation.

Exemple 4. *Par exemple, pour un blog, cela correspond à produire un certains nombres de pages web (disons 100 pages) listant les posts 10 par 10 (items de contenus) grâce à un ensemble de fichiers contenant chacun un post (items de données). Ici, le template est appliqué 100 fois sur des ensembles de 10 posts jusqu'à ce que tous les posts soient traités. Cela génère les 100 pages contenant chacune 10 posts.*

ManyTemplatesIn

Ce cas d'utilisation implique d'utiliser un générateur de contenu statique dans le but de produire un fichier contenant plusieurs fois les mêmes données mais agencées par un *template* différent. Nous n'avons pas trouvé d'exemple pratique pour illustrer ce cas mais la Figure 9 permet de visualiser le format

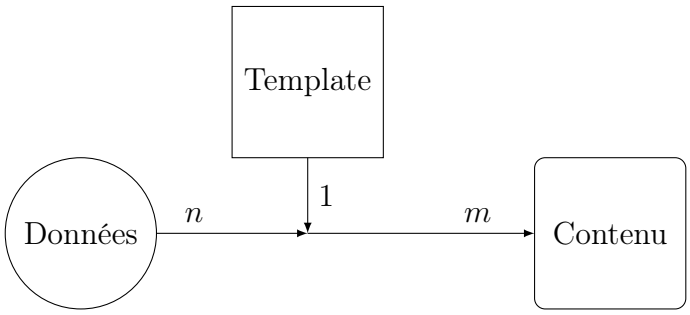

FIGURE 7 – Représentation du cas **ManyToMany**.

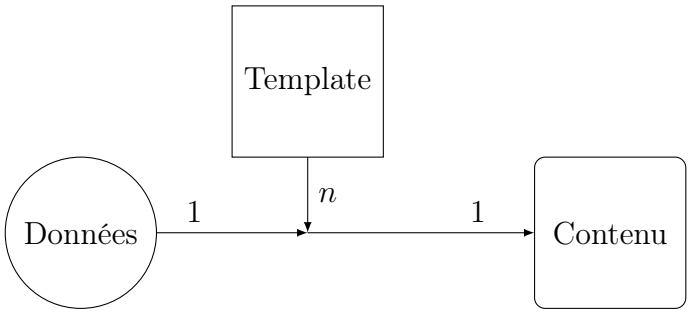

FIGURE 8 – Représentation du cas **ManyTemplatesIn**.

de fichier générer dans ce cas. La Figure 8 exhibe plus clairement les multiplicités de ce cas, où **n** est le nombre de *templates* à appliquer sur le fichier de données.

Remarque 2. *Notre générateur de contenu statique ne permet pas de gérer ce cas directement. Néanmoins, il existe un moyen de contourner cette limitation. En effet, suivant le moteur de template utilisé, il est possible de créer un template qui en inclut d'autres. Cela permet donc de produire un comportement similaire à ManyTemplatesIn.*

ManyTemplatesOut

Ce cas d'utilisation est similaire au précédent, il implique également l'application de plusieurs *templates* sur un même fichier de données. La différence vient du fait que *ManyTemplatesOut* produit un fichier en sortie par *template* appliqué. L'Exemple 5 met ce cas d'utilisation en pratique. Cela se traduit par la Figure 10 où **m** est le nombre de *templates* à appliquer sur le fichier.

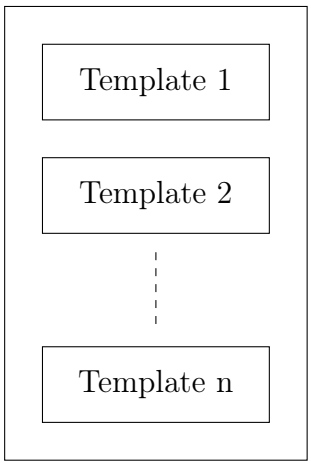

FIGURE 9 – Représentation d'un fichier de sorti du cas **ManyTemplatesIn**.

Exemple 5. *En pratique, cela revient à dire que pour un fichier contenant les informations propres à un CV, on veut générer un fichier LaTeXet une page web. Ces deux contenus contiendraient les mêmes informations mais l'un pourra être imprimé (une fois le PDF produit) et l'autre sera mis à disposition sur Internet.*

Remarque 3. *Notre générateur de contenu statique ne donne pas à l'utilisateur le contrôle sur la multiplicité des templates. Cependant, comme pour le cas ManyTemplatesIn, il existe un moyen de contourner cette limitation afin de pouvoir réaliser le cas ManyTemplatesOut. En effet, intuitivement, ce cas d'utilisation correspond à* $\boldsymbol{m}$ *cas d'utilisation ManyToOne où* $\boldsymbol{n}$ *vaut 1. Ce cas d'utilisation est donc réalisable mais cela demande un travail supplémentaire de la part de l'utilisateur qui devra réaliser plusieurs cas ManyToOne afin d'y parvenir.*

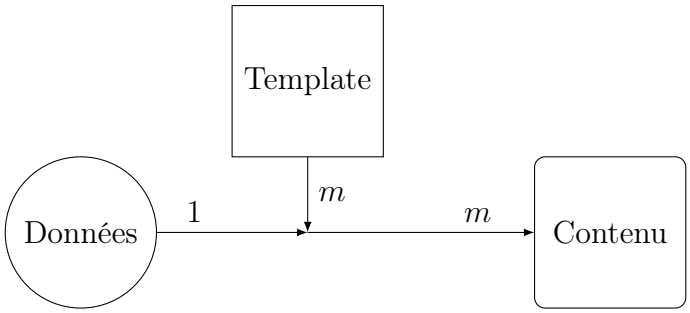

FIGURE 10 – Représentation du cas **ManyTemplatesOut**.

3 Etat de l'art

Cette section détaille, de manière non-exhaustive, l'existence d'autres générateurs de contenu statique similaires à ce projet. Leurs fonctionnalités principales seront présentées afin d'être comparées avec notre propre générateur.

Ces générateurs de contenu statique ont été choisis grâce à StaticGen [1] qui établie un classement d'un grand nombre de ces générateurs sur base de leur popularité sur GitHub. Parmi tous ceux qui y sont listés, nous en avons choisis trois comme étalons pour notre projet. Il s'agit de Jekyll, Pellican et Lektor.

Nous pouvons expliquer ces choix comme suit : Jekyll est le générateur *open-source* le plus populaire. Il s'agit donc d'un bon point de comparaison. Ensuite, Pellican est le générateur le plus populaire écrit en Python. Comme notre projet est également réalisé en Python, nous trouvons qu'il s'agit également d'une comparaison intéressante. Enfin, Lektor est le quatrième générateur le plus populaire réalisé en Python. Il offre cependant une plus grande souplesse d'utilisation que Pellican. Comme la souplesse d'utilisation est un critère important pour notre projet, nous avons jugé bon de l'ajouter à notre comparaison.

Pelican

Pelican [2] est un générateur de site web statique dont les principales fonctionnalités sont :

- Le support de *pages* (e.g. "Contact", ...)
- Le support d'*articles* (e.g. posts d'un blog)
- La régénération rapide de fichiers grâce à un système de caches et d'écriture sélective.
- La gestion de thèmes créés à partir de *templates Jinja2*
- La publication d'articles dans plusieurs langues

Pelican est un générateur de contenu statique spécialisé pour la création de blogs. Cela le rend plus facile à utiliser pour créer de tels contenus mais cela restreint également sa souplesse d'utilisation. En outre, Pellican permet de réaliser les cas d'utilisations suivants :

- **OneToOne** : permet de créer une page de blog à partir d'un fichier de données.

— **ManyToOne** : permet de générer une page d'accueil affichant plusieurs posts à partir d'un même nombre de fichiers de données.
— **OneToMany** : certains plugins permettent de générer plusieurs fichiers de contenu à partir d'un fichier de données. C'est par exemple le cas du plugin *pdf*.
— **ManyToMany** : pagination des différentes publications dans les archives.

Nous voyons ici que Pelican est capable de réaliser les trois principaux cas d'utilisations que nous avons exposé dans la section précédente. Néanmoins, Pelican n'est pas un générateur de contenu statique aussi flexible que l'on pourrait le vouloir. En effet, ces cas d'utilisations sont en fait imposés à l'utilisateur. Par exemple, le seul moyen de réaliser un cas **ManyToMany** est de paginer les publications dans les archives. L'utilisation de Pelican est ainsi limitée à la création de blogs.

Lektor

Lektor [4] est également un générateur de site web statique. Ses principales fonctionnalités sont les suivantes :

— Un système de construction intelligent qui ne reconstruit que le contenu qui a été modifié
— Un outil graphique qui permet la modification de pages sans toucher au code source
— Utilisation de système de *templates Jinja2* pour le rendu du contenu
— Un outil permettant la création relativement simple de sites web multilingues
— La possibilité de créer des modèles de données permettant un meilleur contrôle sur la mise en page

On peut donc en conclure que Lektor a une fonction plus générale que Pelican car il n'est pas orienté vers la création de blogs mais vers la création de sites web statiques en général. Malheureusement, comme pour Pelican, les fonctionnalités qu'il propose sont spécialisées pour faciliter cela en dépit de la généricité de son usage. Lektor permet d'effectuer les cas d'utilisations suivant :

— **OneToOne** : permet de générer une page de contenu à partir d'une page de données.
— **ManyToOne** : permet la génération d'une page contenant plusieurs "onglets" (e.g "Contact", "Accueil", ...) à partir d'un même nombre de fichiers de données.

Nous voyons ici que Lektor est incapable de produire plusieurs fichiers

de contenus à partir d'un unique fichier de données. En effet, pour chaque fichier de données, Lektor génère un et un seul fichier de contenu. Il souffre également du même défaut que Pelican à savoir que ces cas d'utilisations sont imposés à l'utilisateur.

Jekyll

Enfin, Jekyll [5] est le plus connu des générateurs de sites web statiques *open source*. Il est écrit en *Ruby* à la différence de Pelican et Lektor, écrits en *Python*. Voici ses principales fonctionnalités :

- Utilisation de *templates Liquid*
- Support de contenu de type *pages* (e.g "Contact", "Accueil", ...) et *articles* (e.g. posts d'un blog)
- Lancement d'un serveur local pour observer le rendu graphique des fichiers générés

Comme pour Pelican, l'utilisation de Jekyll est orientée pour la création de blogs. Cependant, il peut aussi facilement être utilisé pour générer des sites web statiques de manière plus générale. La grande force de Jekyll vient donc de sa simplicité d'utilisation et de sa relative souplesse d'utilisation. En effet, les cas d'utilisations supportés par Jekyll sont les suivants :

- **OneToOne** : permet de créer un post à partir d'un fichier de données.
- **ManyToOne** : permet de générer une page d'accueil contenant plusieurs onglets ("Contact", "About", ...) à partir d'autant de fichiers de données.

Comme Lektor, Jekyll est incapable de générer plusieurs fichiers de contenu à partir d'un seul fichier de données. Cependant, Jekyll est aussi polyvalent que Lektor malgré son orientation pour la création de blogs.

Intérêt du projet

Malgré leurs différences, ces générateurs de contenu statique présentent certaines similitudes au niveau des cas d'utilisations qu'ils permettent de réaliser. En effet, ils offrent tous la possibilité de réaliser le cas **ManyToOne** et donc aussi le cas **OneToOne**. Pelican est également capable de gérer le cas **OneToMany** pour certains cas particuliers. Nous pouvons donc dire que ces générateurs de contenu statiques, bien que très prisés, présentent une certaine limitation au niveau des cas d'utilisation qu'ils sont capables de réaliser.

Le but de ce projet est donc de mettre au point un générateur de contenu statique qui propose à l'utilisateur un maximum de cas d'utilisation afin de ne pas brider l'utilisateur. L'intérêt de ce projet va donc au-delà de la génération de sites web statiques. En effet nous avons vu que des générateurs moins puissant au niveau de la gestion des cas d'utilisations sont capables de mener cette tâche à bien.

4 Analyse fonctionnelle

Cette section est destinée à justifier les choix de conception de ce projet afin de remplir le cahier des charges. Certains choix seront inspirés directement de solutions existantes listées dans la section précédente.

La section suivante, sera directement liée à celle-ci car elle expliquera comment ces choix seront techniquement mis en œuvre lors de la phase d'implémentation.

4.1 Abstraction des données

Un des points forts de notre générateur de contenu statique est de pouvoir abstraire les données qu'il traite. Grâce à ce mécanisme, l'utilisateur est capable de manipuler une grande diversité de type de donnée sans avoir à adapter ses manipulations. Dit autrement, cela permet à l'utilisateur de travailler de la même manière avec plusieurs types de données différents sans avoir à adapter ses requêtes. Cela donne également la possibilité de manipuler ensemble des données syntaxiquement différentes mais sémantiquement les mêmes. Dans ce cas, pour parler des données traitées, nous utiliserons le mot "**item**" introduit dans la section 2.1.

4.2 Configuration

Un autre point fort de notre générateur de contenu statique est qu'il est facilement configurable. En effet, il est possible de configurer son répertoire de travail ainsi que le moteur de rendu utilisé pour les *templates*. Mais cela reste une fonctionnalité assez classique. L'originalité de notre système de configuration vient du fait qu'il est également possible d'y ajouter des modules *Python 3* en plus des modules de bases que nous proposons. Cela permet à l'utilisateur d'y ajouter des fonctionnalités qu'il aurait lui-même développé pour que notre générateur réponde au mieux à ses besoins.

4.3 Règles de génération

Pour ce qui est de l'utilisation en elle-même de notre générateur de contenu statique, nous avons pensé à un système de règles relativement intuitif. Ces règles sont divisées en plusieurs parties appelées **champs**. Chaque champs (ou ensemble de champs) représente un paramètre du générateur de contenu statique. L'explication du fonctionnement de ces règles est approfondies dans la section suivantes. Ces règles permettent également à l'utilisateur de créer des variables qu'il peut ensuite utilisées dans le *template*. Ces règles

sont contenues dans un fichier qui sera passé en paramètre à notre programme qui les exécutera une par une. Chacune de ces règles symbolisera un des trois cas d'utilisation exposé en Section 2 : *AllToOne, OneToAll* et *AlltoMany*.

4.4 Transformation des données

Une fonctionnalité importante de notre générateur de contenu statique est la possibilité d'appliquer des transformations aux données en cours de traitement. Cela peut par exemple permettre à l'utilisateur de formater certaines données avant de les injecter dans le *template*. Afin de faciliter ce processus pour l'utilisateur, nous mettons aussi à sa disposition des méta-fonctions. Ces fonctions sont au nombre de trois : **map**, **reduce**, **filter**.

Map est une méta-fonction qui permet d'appliquer une fonction à chaque *item* d'un ensemble d'*items*. Le résultat est donc un nouvel ensemble d'*items* dont chaque *item* est le résultat de l'application de la fonction sur un *item* de l'ensemble initial.

Reduce est une méta-fonction qui applique une fonction sur un ensemble d'*items* deux par deux. L'un des deux items passé à cette fonction est le résultat obtenu à partir de la paire d'*items* précédente. Le résultat est donc accumulé jusqu'à ce que la fonction ait été appliquée sur tout les *items*. Le résultat de cette méta-onction est donc un unique *item*.

Filter est une méta-fonction qui applique une fonction booléenne à chaque membre d'un ensemble d'*item*. Le résultat de cette méta-fonction est un nouvel ensemble d'*items* contenant uniquement les *items* dont le résultat de la fonction est `True`.

Ces trois méta-fonctions permettent à l'utilisateur de transformer les données en cours de traitement avec une très grande flexibilité.

En plus de ces méta-fonctions, nous mettons également à disposition de l'utilisateur toute une série de fonctions de base pour l'aider à traiter au mieux ses données.

4.5 Ajout de méta-données

Chaque donnée traitée par notre générateur de contenu statique se voit assigner des méta-données en fonction des différentes transformations que celle-ci subit. Ces méta-données peuvent donc par exemple représenter le nom du fichier duquel les données ont été chargées, l'extension du fichier, ... Ces méta-données sont ensuite utilisable dans le *template* au même titre que les données elles-mêmes.

Syntaxe

Un objectif important de notre générateur de contenu statique est de fournir à l'utilisateur une syntaxe compacte et intuitive. Cela s'applique tout particulièrement aux règles de générations qui est la partie la plus complexe à gérer pour un utilisateur.

D'une part nous n'obligeons l'utilisateur qu'à manipuler des fichiers au format relativement simple, à savoir au format *YAML*. D'autre part, les expressions utilisée dans les règles de générations constituent une grande partie de cette complexité.

Nous avons donc choisi de faire en sorte que le résultat d'une fonction soit automatiquement passée en paramètre à la fonction suivante grâce à l'opérateur "»". Cela permet ainsi de bien séparer chaque appel de fonction au niveau syntaxique. Et donc cela améliore aussi la lisibilité.

De plus, ce comportement est assez intuitif car appliquer des fonctions une à une sur des données dans le but de les "raffiner" est typiquement ce que l'utilisateur souhaite faire.

4.6 Système de fichiers *logs*

Remarque 4. *Cette fonctionnalité n'est pas présente dans notre générateur de contenu statique car elle n'était pas spécifiée dans le cahier des charges. Néanmoins, nous trouvons que celle-ci est intéressante, c'est pourquoi nous la détaillons quand même dans cette sous-section.*

Un système de fichiers *logs* permettrait de faciliter la correction des règles lorsque les fichiers en sorties ne correspondent pas à ce que l'utilisateur attend. Ces fichiers *logs* n'ont comme utilité que de lister l'historique des fonctions appliquées pas à pas pour chaque champ de chaque règle. L'utilisateur n'a donc plus qu'à repérer quelle règle et quel champ n'agit pas comme il le voudrait.

Ce système viserait à atteindre l'objectif de simplicité d'utilisation -et d'aide à l'utilisation dans ce cas-ci- mentionné dans le cahier des charges en Section 2.

5 Analyse technique

Cette section est destinée à détailler les techniques mises en œuvres afin d'implémenter les différentes fonctionnalités décrites dans la section précédente. Nous dresserons également une liste non-exhaustive de différentes idée qui permettraient d'améliorer notre générateur de contenu statique. Le but de cette analyse technique est de montrer le raisonnement qui a conduit à la structure actuelle de notre générateur de contenu statique. Nous n'exposerons donc pas les détails d'implémentations. Si cela vous intéresse, le code source à disposition sur `https://github.com/MaximeDeWolf/DGSC/tree/master/code/src`.

5.1 Règles de génération

Une règle de génération est le moyen que nous mettons à disposition à l' utilisateur pour que celui-ci spécifie à notre générateur de contenu statique pour désigner le comportement que celui-ci doit adopter. Typiquement, une règle représente un cas d'utilisation. Il s'agit donc d'un des points les plus importants du projet. Cette sous-section explique donc en détail les différents éléments qui définissent ces règles ainsi que leur structure.

Le format

Le format des fichiers de règles se doit de posséder deux qualités importantes. La première est d'être facilement compréhensible par l'utilisateur aussi bien lors de la lecture que de l'écriture. La deuxième est de pouvoir être analysé facilement par notre générateur de contenu statique dans le but d'y lire les informations nécessaire à son fonctionnement.

Nous avons donc choisis le format *YAML* pour stocker ces règles. Chaque fichier de règles contient donc une liste de règles. Ces règles contiennent chacune quatre champs qui eux-mêmes contiennent une expression. Ce sont ces expressions qui seront évaluées par notre générateur. La Figure 11 illustre le format type d'un fichier de règles.

Les expressions

Les champs

Ces règles possèdent quatre champs obligatoires : *target*, *template*, *data* et *output*. Comme expliqué dans la précédente section, chacun de ces champs (ou ensemble de champs) permet de contrôler les paramètres de notre générateur. *target* et *output* représentent le contenudu générateur tandis que *template* et

```
#first rule
-   target : expression1
    data :
        key: value1
        key2: value2
    template : template1
    output : path1

#second rule
-   target : expression2
    data :
        key: value
    template : template2
    output: path2

#...
```

FIGURE 11 – Format d'un fichier de règles de notre générateur

data représentent respectivement le *template* et les données. Les particularités et fonctions de chacun de ces champs sont expliqués ci-dessous.

Target

Le champ *target* contient une expression destinée à être lue par notre générateur. Le résultat de cette expression sert à définir la multiplicité du paramètre "contenu" de la règle. Cela définit le nombre de fois que les autres champs de la règles courantes doivent être évalués. A chaque itération de ce processus un *item* différent est stocké dans une variable appelée *current*. Cette variable est ensuite utilisable dans les autres champs de la règle courante.

Template

Ce champs contient une expression destinée à être évaluée par notre générateur de contenu statique. Le résultat de cette expression à désigner le *template* à utiliser sur les données chargées grâce au champ *data*.

La multiplicité de ce paramètre est volontairement bloquée à un. En effet, cela simplifie ainsi l'utilisation de notre générateur de contenu statique bien cela limite également souplesse d'utilisation. Cette limitation nous empêche

de réaliser les cas d'utilisation **ManyTemplatesIn** et **ManyTemplateOut**. Nous avons toutefois montré un moyen de contourner cette limitation dans la Section 2.

Data

Ce champs contient un ensemble de paires clés/valeurs. Chacune de ces valeurs est une expression destinée à être évaluée par notre générateur de contenu statique. Chaque clé de cette ensemble devient une variable dont la valeur est le résultat de son expression évaluée par notre générateur. Ces variables sont par la suite utilisables dans le *template* sélectionné par le champ *template*. La multiplicité de ce champs définit la multiplicité du paramètre "données" du cas d'utilisation.

Output

Ce champ contient une expression destinée à être évaluée par notre générateur de contenu statique. Le résultat de cette expression désigne le chemin vers le fichier où le résultat final de l'application du *template* sur les données doit être écrit. Ce champs influence indirectement la multiplicité de paramètre "contenu" du cas d'utilisation. En effet, même si la multiplicité de ce paramètre est sensée être supérieure à un, fournir à tous ces *items* de contenu le même fichier de sortie entraîne l'écrasement de ceux-ci pour au final ne contenir que le dernier *item* généré.

5.2 Configuration

Le fichier de configuration est stocké au format *YAML* pour garantir à l'utilisateur une simplicité de lecture est d'écriture. Ce format permet également à notre générateur de lire facilement. De plus, ce format permet de hiérarchiser les différentes options. Cela nous permet par exemple de séparer les options propres au *template* de celle propre à notre générateur en lui-même. Ces deux catégories s'appellent respectivement **TEMPLATE** et **PRODUCTION**.

Ce fichier de configuration permet à l'utilisateur de spécifier quelques options à notre générateur de contenu statique. Ces options sont les suivantes : répertoire de travail en entrée, les modules à charger, les *loaders*, le répertoire de *templates* et le moteur de *template*. Les trois première font partie de la catégorie PRODUCTION alors que les deux dernières font partie de la catégorie *TEMPLATE*.

Le répertoire de travail en entrée, représenté par le nom **WORKING_DIR** dans le fichier, spécifie le chemin menant au répertoire contenant les fichiers de données. Cela permet à l'utilisateur de ne pas avoir à écrire de longs chemins de fichier dans le champ *data*. Sa valeur par défaut est le répertoire courant : ".".

Les modules à charger, représentés par le nom **MODULES**, énumère les chemins menant aux modules Python écris par un tiers. Ces modules sont ensuite chargés par notre générateur de contenu statique afin que ceux-ci soient utilisables dans les expressions des règles de génération. Par défaut cette liste contient uniquement tous les modules de bases de notre générateur de contenu statique.

Tous les modules doivent contenir une variable globale "**SHORT_NAME**". Celle-ci sert à prévenir les conflits au cas où deux fonctions porteraient le même nom. Ce *SHORT_NAME* concaténé à un point doit donc précéder le nom de la fonction lorsque celle-ci est appelée. Soient *MODULE*, *M* et *Func*, respectivement un module, son *SHORT_NAME* et une fonction faisant partie de ce module est ne prenant pas d'argument. L'utilisation de cette fonction se fait donc comme suit :

$$M.Func()$$

Les *loaders*, représentés par le nom **LOADERS**, listent les chemins menant aux modules Python écris par un tiers. Ces modules sont ensuite chargés par notre générateur de contenus statique afin que ceux-ci soient utilisables. La fonction **load** utilise ensuite ces modules lorsqu'elle est appelée dans le but de pouvoir charger des fichiers sur base de leur extension. Par défaut, *LOADERS* contient au moins les *loaders* de base c'est à dire ceux destiné à charger des fichiers *YAML* et *Json*.

Le répertoire de *templates*, représenté par le nom **DIR**, est le chemin menant au répertoire contenant les fichiers *template*. Cela permet à l'utilisateur d'éviter d'écrire de trop long chemins de fichier dans les expressions destinées à désigner le chemin d'un *template*. Sa valeur par défaut est ".".

Le moteur de *template*, appelé **BACKEND**, est le chemin du module Python qui permet à notre générateur de contenu d'utiliser un moteur de *template* installé par l'utilisateur. Ce module se doit de respecter une certaine interface pour que cela soit possible. Il doit implémenter les méthodes suivantes : **initEnvironment**, **loadTemplate**, **loadData** et **render**. La méthode *initEnvironment* permet d'initialiser le répertoire de *template* pour le

```
TEMPLATE:
    BACKEND: 'renderers/jinja_renderer.py',
    DIR: '.'

PRODUCTION:
    WORKING_DIR: '.'
    MODULES:
        - 'loaders/loader.py'
        - 'loaders/lister.py'
        - 'loaders/extractor.py'
        - 'transformers/filename_transformer.py'
        - 'transformers/meta_functions.py'
        - 'transformers/data_transformer.py'
        - 'transformers/wrapper.py'
        - 'filters/selection_functions.py']
```

FIGURE 12 – Configuration par défaut de notre générateur

moteur de *template*. Les méthodes *loadTemplate* et *loadData* permettent respectivement de charger un *template* et de charger des *données*. Enfin, *render* permet d'appliquer le *template* chargé sur les données et retourne le résultat final. La valeur par défaut de cette option est "renderers/jinja_renderer.py".

Le chemin d'un fichier de configuration peut être passé en paramètre en ligne de commande lors de l'évaluation d'un ou plusieurs fichiers de règles. Cela permet à l'utilisateur d'avoir un contrôle précis sur la configuration qui est appliquée à un ou plusieurs fichiers de règles.

Si aucun fichier de configuration n'est passé en paramètre à la ligne de commande, notre générateur chargera par défaut la configuration appelée "config.yaml".

Si cette configuration n'existe pas non plus, notre générateur utilisera alors la configuration par défaut. La Figure 12 montre cette configuration par défaut.

Un fichier de configuration n'a pas besoin de spécifier toutes les options. Celles qui ne sont pas mentionnées prennent automatiquement les valeurs par défaut. Si une configuration spécifie les options *MODULES* et *LOADERS* alors ils sont automatiquement agrémentés de leurs valeurs par défauts respectives.

Remarque 5. *En pratique, la configuration par défaut présentée à la Figure*

12 est encodée directement dans un dictionnaire Python. Cependant, nous nous jugeons préférable de vous la montrer au format YAML pour qu'elle serve également à illustrer le format que doit avoir un fichier de configuration.

5.3 Syntaxe

Comme expliqué dans la section précédente, le résultat de chaque fonction est envoyé à la suivante. Pour garantir ce comportement, nous permettons à chaque fonction d'être évaluée partiellement, c'est-à-dire que nous permettons aux fonctions d'être évaluées même si elles ne possèdent pas tous les arguments nécessaires.

Cela vient du fait que ce comportement est induit par l'opérateur "»" et donc les membres de droite et de gauche sont évalués séparément avant d'appliquer l'opérateur. Le problème est que le membre de droite à besoin du résultat du membre de gauche pour être évalué correctement. C'est à ce moment que l'évaluation partielle entre en jeu.

Grâce à celle-ci, le membre de droite retourne une fonction qui pourra être totalement évaluée une fois que le membre de gauche lui enverra son résultat.

De plus, ce processus est totalement transparent pour l'utilisateur ce qui fait que cette technique ne présente que des avantages.

5.4 Abstraction des données

Cette fonctionnalité permet de traiter les données exactement de la même manière indépendamment du type de fichier duquel elles proviennent. Cela est rendu possible grâce aux *loaders* qui ont pour rôle de charger un fichier de données et d'encapsuler celles-ci dans des *items*.

Ces *items* possèdent typiquement une variable *data* qui contient les données chargées sous forme de dictionnaire Python. Ils possèdent également une variable *info* qui stockent les méta-données produites après chaque transformation. Ces méta-données sont elles aussi stockées au format dictionnaire Python.

Cette variable *info* peut être accédée depuis une expression en spécifiant le nom de l'*item* suivi d'un point et du nom de la méta-donnée. Soient *Current* et *datapath*, respectivement un *item* et le nom d'une méta-donnée, la valeur de celle-ci peut être accédée comme suit : *Current.datapath*. La variable *data* n'est quant à elle accessible que grâce à la fonction *E.fetch* qui accède aux données et les ré-encapsule dans un *item*.

Toutes les fonctions mises à disposition de l'utilisateur manipulent en fait des *items* et pas les données en elles-mêmes. Ces fonctions sont chacune responsables d'ajouter les méta-données qui conviennent ainsi que de ré-encapsuler les données dans un *item* le cas échéant.

5.5 Améliorations possibles

Cette sous-section présente des idées qui ont pour but d'améliorer notre générateur de contenu statique. Il s'agit bien sûr d'une liste non-exhaustive.

Performances

Durant le développement de notre générateur, nous ne nous sommes pas intéressé aux performances de celui-ci. Cela pourrait poser problème si notre générateur était utilisé pour un projet de grande envergure.

Une amélioration possible serait de ne re-générer que les fichiers qui auraient subi une modification. Actuellement notre générateur applique toutes les règles une à une et ré-écrit tout les fichiers produits. Il suffirait donc de vérifier si le fichier qui devrait être générer existe déjà et si sa date de dernière modification est plus récente que celle du fichier de règle. Si c'est la cas, alors le fichier n'a pas besoin d'être à nouveau généré.

Fonction de tri

Notre générateur de contenu statique propose actuellement des fonctions de base pour traiter des données. Néanmoins, il n'existe pas de fonction permettant de trier les *items* selon les données qu'ils contiennent. Cela pourrait être une fonctionnalité intéressante.

Plus d'options de configuration

Le mécanisme de configuration de notre générateur ne prend en charge que peu d'options. On pourrait par exemple ajouter un équivalent de l'option "répertoire de *template*" pour les données et les fichiers de sorties.

Cela demande plus travail qu'il n'y parait car pour les *templates* et les fichiers de sorties, il ne suffit que de concaténer la valeur de l'option avec le résultat obtenu dans le champs correspondant. Cela n'est pas possible avec les données car les valeurs manipulées par le champs correspondant est un ensemble d'*items*.

6 Conclusion

Nous avons mis au point un générateur de contenu statique doté d'une grande souplesse d'utilisation. En effet, il différent des autres générateurs de contenu statique de par sa capacité à permettre à l'utilisateur de réaliser une plus grande diversité de cas d'utilisation. Cette particularité lui permet également de ne pas se cantonner à la génération d'un seul type de contenu. Sa polyvalence est donc également une de ses forces.

De plus, nous avons fait notre possible pour proposer à l'utilisateur la syntaxe la plus claire et la plus simple possible. En effet, les fichiers propres à la manipulation de notre générateur, comme les fichiers de configurations et de règles, sont au format *YAML*, un format relativement simple à prendre en main. En plus de cela, nous avons pris soin à ce que la syntaxe des expressions utilisées dans les règles de génération soit la plus claire et la plus intuitive possible.

Notre générateur de contenu statique donne également à l'utilisateur la possibilité de transformer les données chargées à la volée. Pour ce faire, nous mettons à disposition de l'utilisateur un ensemble de fonctions de base ainsi que trois méta-fonctions. Notre générateur donne aussi l'opportunité de traiter toutes sortes de fichiers de données sans devoir modifier les règles de génération. Cela est possible grâce aux *loaders* qui se charge de l'abstraction de ces données.

Enfin, si l'utilisateurs veut avoir accès à d'autres fonctions ou pouvoir charger des données d'un type non pris en charge, il peut écrire lui-même ses modules et ses *loaders*. Il peut ensuite les encoder dans un fichier de configuration afin de pouvoir les utiliser dans les règles de génération.

Même si nous avons tout de même constaté qu'il pouvait être amélioré relativement facilement, notre générateur de contenu statique respecte toute les exigences du cahier des charges et contient toutes les fonctionnalités présentée dans l'analyse fonctionnelle. Nous pouvons donc dire que le générateur que nous avons développé comporte toutes les fonctionnalités désirées à sa création.

Bibliographie

[1] Classement des générateurs de contenu statique *open-source*. https://www.staticgen.com/. (consulté le 19/08/18).

[2] Documentation de pelican. http://docs.getpelican.com/en/stable/. (consulté le 14/11/17).

[3] Page d'accueil de jinja2. http://jinja.pocoo.org/. (consulté le 31/12/17).

[4] Page d'accueil de lektor. https://www.getlektor.com/. (consulté le 14/11/17).

[5] Présentation de jekyll. https://jekyllrb.com/. (consulté le 14/11/17).