Projet 2020-2021
Génie Logiciel Avancé

version 1.0.8

Adrien Guatto & Stefano Zacchiroli

9 avril 2021

Sommaire

1. Introduction

Le génie logiciel est un sujet principalement empirique : ses bases s'acquièrent et s'entretiennent par une pratique assidue de la programmation. C'est pourquoi, dans le cadre du module de génie logiciel avancé, l'équipe enseignante va vous guider dans la réalisation un projet logiciel de relativement grande envergure. Celle-ci servira de véhicule et d'illustration aux concepts et techniques abordées durant le cours magistral :

— programmation objet moderne avec patrons de conception (*design patterns*),
— tests unitaires, tests d'intégration et intégration continue,
— méthodes agiles (e.g., SCRUM) pour le développement en équipe,
— et bien d'autres choses encore.

L'équipe enseignante répondra aux questions posées sur la plateforme en ligne du cours, située à l'adresse

`https://moodle.u-paris.fr/enrol/index.php?id=10699`.

2. Méthodologie

Avant d'entrer dans le cahier des charges du projet à proprement parler, précisons quelques informations sur la méthodologie à suivre pour sa réalisation.

2.1. Organisation

Les projets doivent être réalisés en équipes indépendantes comprenant chacune six étudiants (hexanômes). Leur constitution aura lieu en deux étapes :

1. les étudiants s'organisent librement en trinômes,
2. les enseignants apparient les trinômes aléatoirement pour former les hexanômes.

Les étudiants qui n'ont pas choisi leur trinôme avant la **première séance de travaux pratiques** seront affectés à un hexanôme aléatoire.

Plagiat. Si les échanges d'idées entre équipes sont permis et mêmes encouragés, l'échange de code est lui strictement proscrit. Tout plagiat entraînera l'attribution d'un zéro final aux équipes concernées.

2.2. Qualité du code

Nous attendons de votre projet :

— un code conforme aux exigences de qualité logicielle standard (fiable, commenté judicieusement, facilement extensible),

— des procédures de construction (*build*) et de déploiement documentées, avec l'utilisation des outils standard du domaine,

— une procédure de test automatisée rigoureuse, avec une bonne couverture de votre code ($\geq 70\%$).

Le cours magistral aura vocation à vous donner les concepts et techniques nécessaires pour que votre projet réponde à ces attentes. De plus, le déroulement du projet est structuré de sorte à favoriser naturellement les projets qui obéissent à ces bonnes pratiques.

2.3. Intégration continue

L'*intégration continue* (ou *continuous integration* en anglais) permet d'exercer votre suite de tests automatiquement, notamment à chaque *merge request*. Vous devez utiliser le support de GitLab pour ce faire, via le concept de *runner*, qui est installé sur Gaufre.

https://docs.gitlab.com/runner/

L'utilisation de Docker est fortement conseillée.

Le `README.md` de votre dépôt doit afficher le résultat des tests via des badges colorés. Toute *merge request* sur votre dépôt doit afficher l'intégration continue également.

2.4. Publication continue

La *publication continue* (ou *continuous delivery* en anglais) permet d'automatiser la production des artéfacts logiciels qui résultent de votre travail. Elle permet un processus de publication léger, où une version du logiciel correspond simplement à un *tag* dans son dépôt Git. Lors d'une poussée d'un nouveau *tag*, votre projet doit produire et publier automatiquement l'archive au format zip de la version correspondante. Cette archive doit contenir les fichiers nécessaires à l'exécution du projet, en particulier l'archive jar contenant le bytecode Java et le script shell qui simplifie son exécution. Un onglet *releases* sur GitLab doit permettre de télécharger les archives récentes.

2.5. Initiatives supplémentaires

Les instructions données ci-dessus fournissent une base méthodologique minimale pour renforcer la qualité logicielle de votre projet. Vous êtes fortement incités à mettre en place des procédures supplémentaires. Par exemple, il peut être bénéfique d'intégrer des outils d'analyse statique tels que SonarQube ou Infer à vos processus de développement.

3. Description générale

3.1. Aperçu

Le but du projet est la réalisation d'un *générateur de sites web statique* écrit en langage Java. Il s'agit d'un outil capable de traduire un ensemble de fichiers écrits dans un format de document de haut niveau (type *Markdown*) vers HTML, CSS et éventuellement JavaScript. Parmi les générateurs de sites web statiques les plus connus, citons Jekyll [6], Hugo [5] ou, dans un genre un peu différent, Ikiwiki [2].

Contrairement aux logiciels de gestion de contenu dynamiques comme Wordpress, MediaWiki ou Joomla, un générateur de site statique n'exige pas du serveur web qu'il exécute du code à la volée, ce qui a de nombreux avantages en termes de performance et sécurité. En revanche, une telle solution est bien entendu moins flexible, puisqu'elle ne permet a priori pas l'édition des fichiers en ligne.

3.2. Déroulement

Le projet sera réalisé en deux phases, qui consiste à la réalisation d'un projet de base puis à son extension. La section 4 spécifie le travail à réaliser pour chacune d'entre elles.

Phase I. La première phase occupera le premier tiers du semestre. Elle consiste à programmer un outil minimal, doté uniquement de fonctionnalités de base, qui servira de socle. Les fonctionnalités à réaliser sont ici imposées et les mêmes pour tous les groupes.

Phase II. La phase d'extension occupera le reste du semestre. Nous vous proposerons une batterie de fonctionnalités parmi lesquelles chaque groupe pourra choisir à sa guise.

3.3. Technologies

Votre code source doit être écrit en langage Java, version 14+.

Il doit compiler et gérer ses dépendances avec Gradle [1]. Vous pouvez utiliser toute bibliothèque externe approuvée par l'équipe enseignante, en plus de celles offertes en standard dans le *Java Development Kit*. La liste des bibliothèques externes actuellement approuvées constitue l'appendice B du présent document.

3.4. Interface utilisateur

Votre projet doit fonctionner en ligne de commande. Bien que Java soit portable, vous fournirez un script shell permettant d'appeler facilement votre code (e.g., passant les bons arguments à la JVM) depuis un système GNU/Linux. Ce script doit s'appeler `ssg`.

L'interface utilisateur de ligne de commande de `ssg` doit être structurée à la manière de celle de Git, sous la forme `ssg COMMANDE [OPTIONS] [ARGUMENTS]`, où `[OPTIONS]` est une liste d'options préfixées par `--` qui dépend de la commande choisie et `[ARGUMENTS]` est une liste d'arguments.

Abbréviation	Résumé	Difficulté	Description
md2html	Traduction simple de fichiers Markdown en fichiers HTML5.	⋆⋆	§4.2.1
buildsite	Gestion de plusieurs fichiers organisés en hiérarchie, et fichier de configuration global.	⋆	§4.2.2
help	Affichage d'un message d'aide.	⋆	§4.2.3
metadata	Métadonnées.	⋆⋆	§4.2.4
static	Fichiers statiques.	⋆	§4.2.5
templates	Patrons et substitution.	⋆⋆⋆	§4.2.6
incremental	Traduction avec recalcul minimal.	⋆⋆⋆⋆	§4.2.7
parallel	Traduction parallèle.	⋆⋆⋆⋆	§4.2.8
http	Serveur HTTP pour le développement.	⋆⋆	§4.2.9
online-edit	Édition en ligne.	⋆⋆⋆	§4.2.10
watchdog	Recompilation en cas de changement.	⋆⋆⋆	§4.2.11
newmarkup	Nouveau format de balisage en entrée.	⋆/⋆⋆	§4.2.12
themes	Thèmes partagés.	⋆⋆	§4.2.13
templates++	Patrons étendus.	⋆⋆⋆	§4.2.14
syndication	Syndication.	⋆⋆	§4.2.15
structured	Gestion des données structurées dans le langage de motifs. (**obligatoire**)	⋆⋆	§4.2.16
anything	Extension libre, au choix des étudiants.	⋆/⋆⋆/⋆⋆⋆	§4.2.17

Table 1 – Fonctionnalités

Nous vous préciserons, pour chaque fonctionnalité attendue, quelles commandes sont affectées et les options et arguments minimaux requis.

4. Fonctionnalités

4.1. Généralités

Le projet est découpé en fonctionnalités. Celles qui relèvent de la phase I du projet sont obligatoires, tandis que la phase II vous laisse libre de vos choix. Vous devez cependant garder à l'esprit deux points : certaines extensions sont plus difficiles à réaliser que d'autres, et seront donc valorisées ; il existe des dépendances entre extensions, certaines ne pouvant être réalisées raisonnablement qu'après certaines autres. Nous préciserons dans chaque cas la difficulté estimée, de ⋆ (facile) à ⋆⋆⋆ (difficile), ainsi que d'éventuels prérequis. Les fonctionnalités sont listées à la table 1, leurs prérequis synthétisés à la fig. 1 (rouge et bleu pour les fonctionnalités de phase I et II).

Recette. Vous devez terminer une fonctionnalité avant de passer à la suivante, et votre réalisation doit être visée par l'équipe enseignante. Chaque extension doit venir avec une

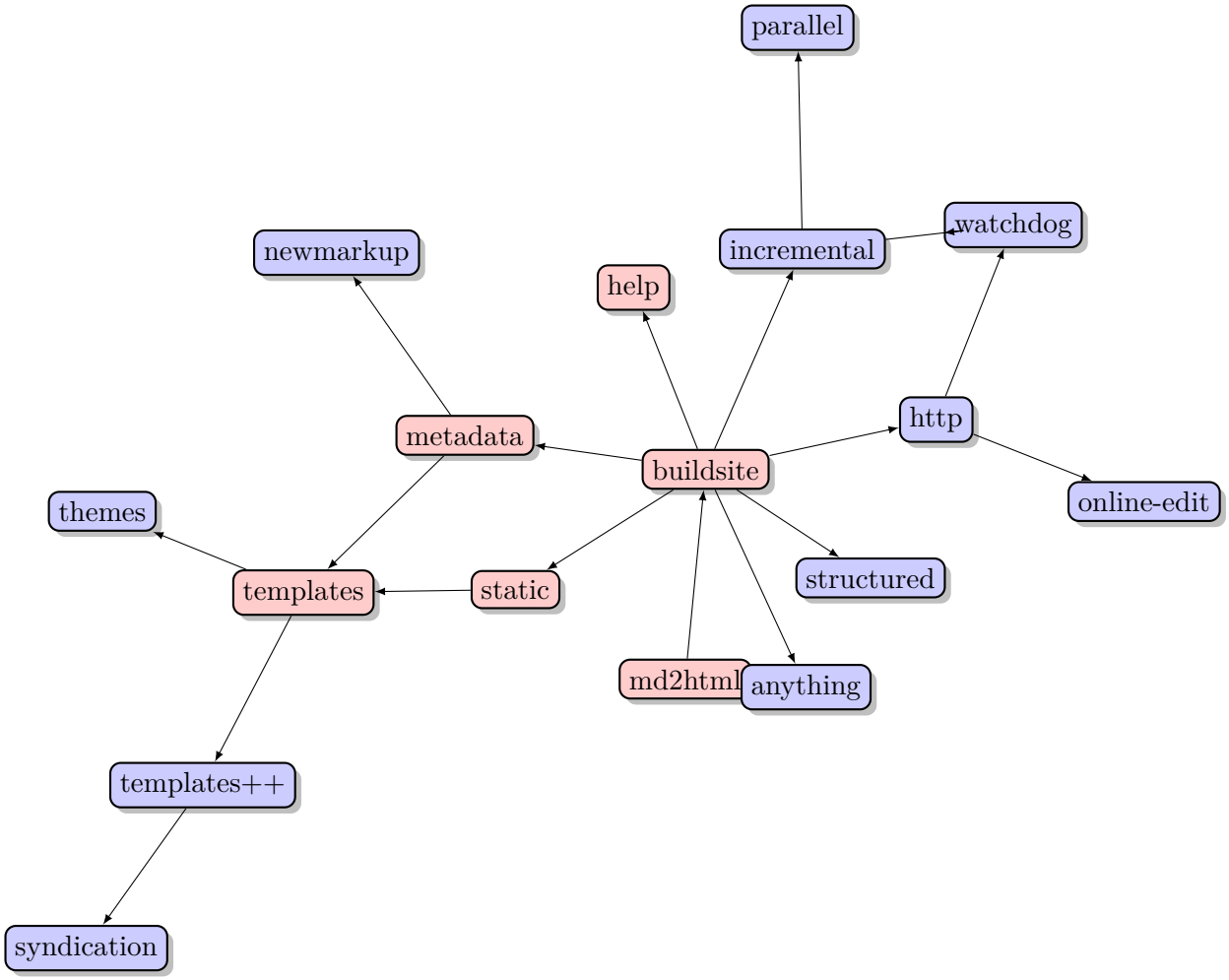

FIGURE 1 – Graphe des dépendances entre fonctionnalités

procédure de test automatisée, et la documentation nécessaire à son utilisation.

4.2. Description des extensions

4.2.1. Traduction simple de Markdown vers HTML

Cette fonctionnalité est la première à implémenter. Elle consiste à traduire un fichier Markdown vers un fichier HTML minimal. Vous devez utiliser la variante CommonMark [3] de Markdown, qui a l'avantage d'être clairement spécifiée. Vous devez générer des fichiers HTML5 corrects et conformes aux standards du web. Votre procédure de test doit tester leur validité, par exemple en utilisant le validateur vNu [7] du *World Wide Web Consortium*.

Interface utilisateur. On doit pouvoir traduire un fichier `file.md` vers `file.html` en appelant `ssg build file.html`. Autrement dit, on ajoute la commande `build`, dont chacun des arguments est un fichier à produire. Les fichiers sont produits dans le répertoire `_output/` par défaut, ou dans le répertoire `DIR` si spécifié via l'option `--output-dir DIR`. Le répertoire destination est créé le cas échéant.

4.2.2. Traduction d'un site complet

La fonctionnalité 4.2.1 ne permet que de construire une page à la fois, pas de gérer un site entier dont les pages partagent certaines caractéristiques et peuvent faire référence les unes aux autres. Votre générateur doit traduire un tel site, stocké sous la forme d'une arborescence complète sur le système de fichier, en une autre arborescence.

On supposera qu'un site suit une arborescence bien particulière. Son format vous est imposé pour la base du projet, et doit contenir les éléments suivants.

— À la racine, un fichier de configuration `site.toml` utilisant le format TOML [4]. L'utilisation du format TOML le rend facilement extensible.

— Dans le sous-répertoire `contents`, un ensemble de fichiers `.md` (cf. §4.2.1) contenant les documents à convertir en pages web, dont au moins un fichier `index.md` qui servira de racine au site.

L'équipe enseignante vous fournira une batterie d'exemples vous permettant de tester votre projet initial, et qui précisera la format attendu.

Interface utilisateur. On enrichit la commande `build` de la fonctionnalité précédente. Invoquée sans arguments, cette commande suppose que le répertoire courant contient une arborescence à traduire. L'argument optionnel `--input-dir` permet de spécifier une arborescence à traiter différente du répertoire courant. Le résultat est stocké dans le sous-répertoire `_output`, sauf si un autre a été spécifié via `--output-dir`. Le listing shell ci-dessous vous montre un exemple d'utilisation.

```
$ cd path_to_website
$ tree
.
```

```
|-- content
|   |-- CommonMark.md
|   |-- index.md
|-- site.toml

1 directory, 3 files
$ ssg build
$ ls _output/
index.html CommonMark.html
```

4.2.3. Aide

Votre logiciel doit être capable de documenter ses entrées en affichant un message d'aide sur la sortie standard. Ce message doit être maintenu avec son évolution.

Interface utilisateur. L'aide doit être accessible via `ssg help`.

4.2.4. Métadonnées

Vous devez étendre votre analyseur de fichiers Markdown avec la possibilité de spécifier des métadonnées. Celles-ci sont spécifiées via la syntaxe TOML en début de fichier, entre deux lignes contenant uniquement `+++`, comme dans l'exemple ci-dessous.

```
$ cat content/souvenirs.md
+++
title = "Souvenirs et aventures de ma vie"
date = "1903-02-13"
author = "Louise Michel"
+++

Lorem *ipsum* dolor sit amet.
```

A priori, un fichier peut contenir des métadonnées TOML quelconques. Toutefois, certaines clés ont une signification standard, et vous devez vérifier que la valeur associée appartient au type spécifié par la table ci-dessous.

Clé	Type de la valeur	Description
`title`	Chaîne de caractères	Titre de la page.
`date`	Date [1] (chaîne de caractères)	Date de rédaction de la page.
`draft`	Booléen	La page est un brouillon.

Comme exemple d'utilisation, votre outil doit ignorer tout fichier Markdown dont les métadonnées comprennent un champ `draft` dont la valeur est *`true`*.

4.2.5. Fichiers statiques

Il est utile de pouvoir ajouter à votre site des fichiers statiques, c'est à dire qui n'ont pas été générés par votre outil. Pensez à des images, des fichiers audio ou vidéo, des feuilles de style CSS, etc. Ces fichiers ne sont pas traités par votre préprocesseur.

Interface utilisateur Un site utilisant votre outil peut désormais contenir un répertoire `static` qui contient des fichiers à copier tels quels dans le sous-répertoire du même nom du répertoire destination.

4.2.6. Patrons et substitution

Traduire chaque fichier Markdown indépendamment vers un fichier HTML autonome n'est pas suffisant pour réaliser un site intéressant. On voudrait par exemple pouvoir inclure un menu de navigation dans tous les fichiers HTMLs générés. Une façon générale d'obtenir ce genre de possibilité consiste à implémenter un moteur de *patrons* (*templates* en anglais).

Chaque fichier Markdown peut désormais spécifier un patron, qui est un fichier texte (par exemple HTML) contenant un ensemble de *motifs*. À chaque motif correspond un *fragment* de texte (par exemple HTML) qui lui est subtitué pour produire le fichier texte final. Pour l'instant, on va se restreindre au petit langage de motifs spécifié par la table suivante.

Motif	Fragment
`content`	Résultat de la traduction en HTML du fichier Markdown.
`metadata.`K	Valeur associée à la clé K des métadonnées.
`include` P	Contenu post-substitution du patron situé au chemin P.

On choisira une traduction raisonnable des valeurs TOML en code HTML, par exemple les listes en utilisant la balise `<li>`, etc.

Un patron est donc un fichier texte contenant des motifs entre double accolades. Chaque fichier Markdown peut spécifier le chemin vers le patron à utiliser par l'intermédiaire de ses métadonnées, via la clé `template`. Si cette clé est absente, on utilise par convention le patron `default.html` s'il existe.

```
$ cat content/index.md
+++
title = "Welcome to Mini-Templates"
date = 2021-01-13
+++

# Index

This is the home page of the Mini-Templates site.
$ cat template/default.html
<html>
```

```
  <head>
    <title>{{ metadata.title }}</title>
  </head>
  <body>
    <div id="menu">{{ include "menu.html" }}</div>
    <h1>{{ metadata.title }}</h1>
    <h2>{{ metadata.date }}</h2>
    <div id="article">{{ content }}</div>
  </body>
</html>
```

Interface utilisateur. Les patrons doivent être stockés dans le répertoire `templates` du site. Tous les chemins vers les patrons sont toujours relatifs au répertoire `templates`. L'exemple `examples/mini-templates` fournit un exemple d'utilisation.

4.2.7. Traduction incrémentale

Faites en sorte que les fichiers cible ne soient regénérés que lorsque les fichiers sources dont ils dépendent ont été modifiés. Attention : ce n'est pas si facile, puisque la notion de dépendance est transitive. Par exemple, tout fichier généré dépend transitivement de `site.toml`.

Interface utilisateur. Inchangée, à l'exception d'une option `--rebuild-all` ajoutée au verbe `build`, qui désactive l'incrémentalité. Si votre implémentation est correcte, la seule différence entre une invocation de votre générateur statique avec et sans l'option `--rebuild-all` est le temps nécessaire à son exécution !

4.2.8. Traduction parallèle

Les processeurs modernes disposent de multiples cœurs capables de travailler simultanément. Tirez-en partie en faisant en sorte que votre générateur traduise simultanément les parties indépendantes du site.

Interface utilisateur. Inchangée, à l'exception d'une option `--jobs` ajoutée au verbe `build`, qui prend un argument entier N déterminant le nombre de fils d'exécution alloués à la traduction. Par défaut, on allouera autant de fils d'exécution que la machine a de cœurs.

4.2.9. Serveur HTTP de développement

Il est utile que votre générateur de site dispose d'un serveur HTTP interne pour pouvoir facilement observer le résultat de la compilation. Autrement, il faut ou bien lire les fichiers directement sur le disque depuis son navigateur, ou bien installer et configurer un serveur HTTP standard (par exemple NGINX ou Lighttpd). (Ce serveur n'a bien sûr pas vocation à être utilisé en production !)

Interface utilisateur. Une commande `serve` qui compile le site puis lance un serveur HTTP sur le port 8080 par défaut. Elle peut accepter les mêmes arguments que `build`, avec en plus un argument `--port` spécifiant un autre port que 8080. Le serveur HTTP expose les fichiers du site compilé.

4.2.10. Éditions en ligne

Enrichissez votre serveur HTTP pour que celui-ci offre une interface d'édition des fichiers. On pourra se contenter d'offrir un éditeur en texte brut dans un premier temps, avant d'aller vers une interface plus riche offrant un aperçu en temps-réel du rendu du code Markdown, voire une interface WYSIWIG.

Interface utilisateur. Un bouton permettant de passer en mode édition sur chaque page du site lorsqu'elle est visualisée avec la commande `serve`.

4.2.11. Recompilation automatique

Il est laborieux d'avoir à relancer la compilation du site durant une phase d'édition. Votre outil doit permettre de recompiler à la volée en cas de besoin.

Interface utilisateur. Une option `--watch` à la commande `build`, qui surveille les fichiers sources et recompile automatiquement le site à chaque modification, avec recalcul minimal. Ce comportement est intégré par défaut à la commande `serve`.

4.2.12. Nouveau format d'entrée

Si Markdown est sans doute le format de balisage léger le plus populaire, il est loin d'être unique. Vous pouvez ajouter à votre générateur de site le support pour d'autres formats. N'oubliez pas d'ajouter le support des métadonnées si nécessaire. Voir la page wikipédia *Lightweight markup language* pour une liste volumineuse. Demandez l'approbation de l'équipe enseignante avant de choisir un format dans la liste.

Interface utilisateur. On doit pouvoir ajouter des fichiers aux extensions autres que `.md` à votre dossier `content`.

4.2.13. Thèmes

Pour permettre la réutilisation de l'apparence d'un site, il est utile de regrouper les patrons et les fichiers statiques (notamment les feuilles de style CSS) dans une unité conceptuelle appelée *thème*, facile à redistribuer.

Un thème est un sous-répertoire du répertoire *themes*. Il peut lui-même contenir un répertoire `static` et un répertoire `templates`, qui fournissent des fichiers utilisables par le reste du site comme s'ils étaient dans les répertoires `static` et `templates` racine du site. Si un patron ou fichier statique est présent à la fois à la racine du site et dans le thème courant, c'est la version à la racine du site qui est préférée.

Interface utilisateur Le thème utilisé par le site est spécifié par le fichier `site.toml`. Le nom du thème est par définition le nom du répertoire correspondant dans le dossier `themes`.

4.2.14. Patrons étendus

Vous pouvez proposer une extension de l'expressivité du langage de motifs, par exemple en vous inspirant du système Jinja 2 :

https://jinja.palletsprojects.com/en/2.11.x/.

En particulier, la possibilité de parcourir des types de données structurés TOML ouvre de nouvelles perspectives. Par exemple, une page servant d'index à un blog peut spécifier dans ses métadonnées la liste de ses billets, et votre moteur doit être capable de générer un index HTML à partir de celle-ci.

4.2.15. Syndication

Les flux RSS et Atom offrent une façon commode d'avertir les visiteurs d'un site de la présence de contenu nouveau. Votre moteur de motifs peut être étendu pour permettre la substitution à l'intérieur d'un fichier d'extension `.rss` ou `.atom`. Couplé avec les fonctionnalités décrites à la section 4.2.14, la gestion de flux devrait rendre votre générateur de site très adapté à la génération de blogs personnels.

4.2.16. Gestion des données structurées dans le langage de motifs

Cette fonctionnalité est obligatoire.

Jusqu'ici, les métadonnées injectées dans vos patrons n'ont été que des données scalaires : chaînes de caractères, dates, nombres, etc. La réalisation de sites web riches requiert un langage de motifs capable de parcourir des données structurées : listes, dictionnaires, etc. Par exemple, comme mentionné à la section 4.2.14, un blog doit être capable d'itérer sur la liste des billets pour l'afficher sous forme de menu.

Pour implémenter cette fonctionnalité, votre projet doit *obligatoirement* offrir les deux possibilités suivantes :

1. une construction d'itération sur les éléments d'une liste via une construction `for ... in ...` similaire à la construction qu'offre Jinja 2 ;
2. une primitive `list_files(path, rec)` renvoyant les fichiers contenu dans le répertoire `path` sous la forme d'une liste de chaînes de caractères corrsepond à leurs chemins relatifs depuis `path`. Le chemin `path` est relatif à la racine du répertoire `content` du site en cours de construction. Si le booléen `rec` est vrai, la fonction parcourt récursivement tous les sous-répertoires de `path`, et renvoie la liste des chemins des fichiers (pas des répertoires) dans l'ordre profondeur-d'abord.

Si vous avez réalisé la fonctionnalité présentée à la section 4.2.14, vous avez sans doute déjà implémenté le premier point. Le second est illustré par l'exemple ci-dessous.

```
{% for post in list_files("blog", false) %}
  <li>{{ post }}</li>
{% endfor %}
```

Pour aller plus loin. Ces deux poins sont un minimum, vous pouvez tout à fait implémenter des constructions plus avancées, en lien avec la fonctionnalité *templates++*. Par exemple, l'extrait d'un patron `landing.html` suivant

```
{% for category in metadata.categories %}
  <li><a href="{{ category.url }}">{{ category.name }}</a></li>
{% endfor %}
```

sera utilisé avec le fichier Markdown

```
+++
title = "La Conquête du pain - Boulangerie"
author = "Pierre Kropotkine"
categories = [{ name = "Accueil", url = "index.html" },
              { name = "Nos pains", url = "pains/" },
              { name = "À propos", url = "àpropos.html" }]
template = "landing.html"
+++

Bienvenue sur le site de la boulangerie _La Conquête du pain_.
```

pour produire un site contenant un menu facilement extensible.

4.2.17. Extension libre

Vous pouvez tout à fait proposer à l'équipe enseignante une extension qui ne figure pas dans la liste précédente. Si elle est acceptée, nous vous préciserons sa difficulté estimée et d'éventuels prérequis.

A. Références

[1] Hans Dockter et al. *Gradle*. Version 6.7.1. 16 nov. 2020. URL : https://gradle.org.

[2] Joey Hess. *Ikiwiki*. Version 3.20200202.3. 2 fév. 2020. URL : https://ikiwiki.info.

[3] John MacFarlane. *CommonMark Spec v0.29*. Rapp. tech. URL : https://spec.commonmark.org/0.29/.

[4] Tom Preston-Werner, Pradyun Gedam et al. *Tom's Obvious Markup Language*. Version 1.0.0. 13 jan. 2020. URL : https://toml.io/en/v1.0.0.

[5] The Hugo Authors. *Hugo*. Version 0.79.0. 27 nov. 2020. URL : https://gohugo.io.

[6] The Jekyll Team. *Jekyll*. Version 4.2.0. 14 déc. 2020. URL : https://jekyllrb.com.

[7] World Wide Web Consortium. *Nu HTML Checker (v. Nu)*. Version 20.6.30. 30 juin 2020. URL : https://validator.github.io/validator/.

B. Bibliothèques approuvées

Les bibliothèques suivantes ont été approuvées par l'équipe enseignante et peuvent donc être utilisées librement dans votre projet :

Bibliothèque	Page d'accueil
TomlJ	https://github.com/tomlj/tomlj
Netty	https://netty.io
CommonMark-java	https://github.com/commonmark/commonmark-java
Guava	https://github.com/google/guava
Picocli	https://github.com/remkop/picocli
jinjava	https://github.com/HubSpot/jinjava
Jackson et ses sous-bibliothèques	https://github.com/FasterXML/jackson
Lombok	https://projectlombok.org/
AsciiDoctorJ	https://docs.asciidoctor.org/asciidoctorj/latest/

Nous vous rappelons que vous devez utiliser Gradle pour compiler votre projet et gérer ses dépendances. Pour être approuvée, une bibliothèque doit être présente dans le dépôt JCenter [2] ou Maven Central [3].

Avertissement : ne vous sentez **pas** obligés d'utiliser les bibliothèques ci-dessus. Certaines ne sont pertinentes que pour certaines fonctionnalités, d'autres peuvent être complètement évitées.

C. Versions de ce document

17/01/2021 Version 1.0.

2. https://bintray.com/bintray/jcenter
3. https://search.maven.org

29/01/2021 Version 1.0.1, bibliothèque `CommonMark-java` autorisée.

12/02/2021 Version 1.0.2, bibliothèques `Guava`, `Picocli` et `JUnit` autorisées. Correction d'une phrase incorrecte en section 4.2.1 (on ne génère que du HTML).

02/04/2021 Version 1.0.3, bibliothèques `Jinjava`, `Jackson`, `Lombok` et `AsciiDoctorJ` autorisées.

02/04/2021 Version 1.0.4, ajout d'une fonctionnalité section 4.2.15 à part.

09/04/2021 Version 1.0.5, ajout d'une fonctionnalité section 4.2.16 à part.

09/04/2021 Version 1.0.6, ajout de précisions sur le résultat de `get_path`.

09/04/2021 Version 1.0.7, mise en cohérence des noms de clefs dans section 4.2.16.

09/04/2021 Version 1.0.8, précision dans section 4.2.16.