Berney Alec, Forestier Quentin, Herzig Melvyn

GEN – 11.06.2021

Portfolio

GÉnie LOgiciel



Table des matières

[Introduction 4](#_Toc74322376)

[Outils 4](#_Toc74322377)

[Questions préalables 4](#_Toc74322378)

[Pourquoi signer les commits avec GPG 4](#_Toc74322379)

[Commit ealry, commit often 4](#_Toc74322380)

[Que doit contenir un message de commit ? 4](#_Toc74322381)

[Conventions de codage 5](#_Toc74322382)

[Langues 5](#_Toc74322383)

[Accolades 5](#_Toc74322384)

[Entête de fichier 5](#_Toc74322385)

[Commentaires 5](#_Toc74322386)

[Classes 5](#_Toc74322387)

[Méthodes 5](#_Toc74322388)

[Nommage 5](#_Toc74322389)

[Workflow 6](#_Toc74322390)

[Processus piloté ou agile 6](#_Toc74322391)

[Collaboration 6](#_Toc74322392)

[Branches 6](#_Toc74322393)

[Issues 6](#_Toc74322394)

[KanPan 7](#_Toc74322395)

[Commit 7](#_Toc74322396)

[Intégration continue 7](#_Toc74322397)

[Description du projet 8](#_Toc74322398)

[Sprint 1 9](#_Toc74322399)

[Réunion de début de sprint 9](#_Toc74322400)

[Conception 9](#_Toc74322401)

[User requirement 9](#_Toc74322402)

[Use cases 9](#_Toc74322403)

[Collaboration 9](#_Toc74322404)

[Stories 9](#_Toc74322405)

[Saisie des données structurées 9](#_Toc74322406)

[Saisie du contenu 10](#_Toc74322407)

[Format des pages 10](#_Toc74322408)

[Affichage de la version 10](#_Toc74322409)

[Initialiser un site statique 10](#_Toc74322410)

[Compiler un site statique 10](#_Toc74322411)

[Nettoyer un site statique 10](#_Toc74322412)

[Amélioration de l’intégration continue 11](#_Toc74322413)

[Release et documentation 11](#_Toc74322414)

[Tâches 11](#_Toc74322415)

[Code reuse 12](#_Toc74322416)

[Choix de la technologie pour la saisie de contenu 12](#_Toc74322417)

[Choix de la technologie pour les fichiers de configuration 13](#_Toc74322418)

[Test-first programming 14](#_Toc74322419)

[Commit early, commit often 14](#_Toc74322420)

[Réunion de fin de sprint 14](#_Toc74322421)

[Sprint 2 15](#_Toc74322422)

[Réunion de début de sprint 15](#_Toc74322423)

[Choix du moteur de template 15](#_Toc74322424)

[Conception incrémentale et décomposition 16](#_Toc74322425)

[Diagramme de classe 16](#_Toc74322426)

[Diagramme de séquence 16](#_Toc74322427)

[Stories 17](#_Toc74322428)

[Moteur de template 17](#_Toc74322429)

[Serveur HTTP 17](#_Toc74322430)

[Continuous delivery 17](#_Toc74322431)

[Tâches 18](#_Toc74322432)

[Refactoring 19](#_Toc74322433)

[Tests d’intégration et tests système 19](#_Toc74322434)

[Automatisation 20](#_Toc74322435)

[Commit early, commit often 20](#_Toc74322436)

[Réunion de fin de sprint 20](#_Toc74322437)

[Sprint 3 21](#_Toc74322438)

[Réunion de début de sprint 21](#_Toc74322439)

[Conception incrémentale et refactoring 21](#_Toc74322440)

[Stories 22](#_Toc74322441)

[-Génération du site statique à la volée 22](#_Toc74322442)

[Manuel utilisateur 23](#_Toc74322443)

[Javadoc 23](#_Toc74322444)

[Code coverage 23](#_Toc74322445)

[Package gen.command 23](#_Toc74322446)

[Package gen 25](#_Toc74322447)

[Code quality 26](#_Toc74322448)

[Code benchmarking 26](#_Toc74322449)

[Tâches 28](#_Toc74322450)

[Refactoring 30](#_Toc74322451)

[Tests d’intégration et tests système 30](#_Toc74322452)

[Automatisation 31](#_Toc74322453)

[Commit early, commit oftern 31](#_Toc74322454)

[Réunion de fin de sprint 31](#_Toc74322455)

[Bilan 31](#_Toc74322456)

# Introduction

Ce document a été réalisé dans le cadre du cours de génie logiciel de la HEIG-VD. Durant ce cours nous avons dirigé un petit projet. Le but de ces pages est de marquer les décisions que nous avons dû prendre au cours de ce travail ainsi que les réponses à différentes questions.

# Outils

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Langage : | Java | IDE : | IntelliJ | Tests : | Maven/JUnit |
| Additionnel : | PicoCLI | **Versionning :** | Git/GitHub | **Déploiement :** | Maven Shade |

# Questions préalables

## Pourquoi signer les commits avec GPG

Grâce à la signature GPG, la source des commits peut être authentifiée. De cette façon, dans certains projets critiques, il est possible de s’assurer de la provenance des modifications du code.

## Commit ealry, commit often

Cette philosophie de commit est bénéfique sous plusieurs aspects. Elle permet d’identifier facilement dans quel commit une fonctionnalité a été implémentée. De cette manière, il est plus facile de revenir en arrière puisque chaque commit implique une quantité limitée de code. De plus, lors de travaux collaboratifs, il est plus facile d’intégrer plusieurs petits commits ensemble que de devoir débugger une grosse intégration multifonctionnelle.

## Que doit contenir un message de commit ?

Ce que l’on souhaite voir dans un commit est subjectif. Toutefois, nous pouvons trouver quelques lignes directrices.

Le titre doit résumer en quelques caractères qu’est-ce qui a été modifié, ajouté ou supprimé.

Le corps du commit sert à expliquer en quelques mots ce qui a été impacté et éventuellement pourquoi, mais pas comment. La donnée comment est obtenue en lisant les différences.

Il faut être précis, mais concis de manière à obtenir rapidement et précisément ce à quoi chaque commit se rapporte.

# Conventions de codage

## Langues

* Code -> En anglais
* Commentaires -> En Français

## Accolades

Toujours après un retour à la ligne

if(...)

{

/\* Du code \*/

}

## Entête de fichier

/\*

-----------------------------------------------------------------------------------

Cours : Génie logiciel (GEN)

Fichier : <NomDuFichier>.h

Auteur(s) : Berney Alec & Forestier Quentin & Melvyn Herzig

Date : JJ.MM.AAAA

-----------------------------------------------------------------------------------

\*/

## Commentaires

Les classes et les méthodes sont commentées au format Javadoc.

### Classes

/\*\*

\* Description de la classe

\* @author auteur 1

\* @date jj-mm-aaaa

\*/

### Méthodes

/\*\*

\* Description de la méthode

\* @param param1 Description du premier paramètre.

\* @param param2 Description du second paramètre.

\* @return Que retourne la méthode..

\*/

### Nommage

Tous les noms seront en camel case sauf pour les constantes où les espaces seront des ‘\_’ et en majuscules.  
**Classe** : Première lettre en majuscule, MaClasse.java  
**Fonction** : Première lettre en minuscule, maFonction()  
**Ma constante** : final int UNE\_CONSTANTE.

# Workflow

## Processus piloté ou agile

Nous avons décidé de choisir un processus agile.

Nous n’avons pas choisi le processus piloté pour une raison principalement. Nous ne sommes pas encore des experts en gestion de projet. De ce fait, il est obligatoire à un moment qu’une tâche à laquelle nous n’avions pas pensé fasse surface ou que nos tâches soient mal agencées. En d’autres termes, nous sentons que nous n’avons pas assez d’expérience pour choisir ce type de pilotage qui laisse peu de marge de manœuvre. Notre manque d’expérience pose également un problème au niveau de l’estimation du temps à passer sur ces chacune des tâches.

Au contraire, nous préférons choisir un processus agile. Il nous permettra d’avancer et d’améliorer notre projet itération après itération. Nous trois, avons pour habitude de programmer de manière à obtenir rapidement un MVP (Minimum Viable Product) que nous améliorons par la suite avec les différentes spécifications du projet. Ainsi en optant pour cette méthode de travail nous serons plus aptes à faire face aux imprévus et aux changements en cas de besoin.

## Collaboration

L’outil principal pour la collaboration sera GitHub. <https://github.com/gen-classroom/projet-berney_forestier_herzig>

### Branches

Chaque itération aura sa branche dédiée appelée sprintx où x est le numéro de l’itération. Puis, dans les branches d’itérations, nous aurons des sous-branches qui serviront à l’implémentation des différentes fonctionnalités au sein de l’itération. Puis à la fin d’une itération, la branche sera merge sur main. Le nom des branches est en anglais.

Le nom des branches doit être écrit en minuscules et les espaces doivent être remplacés par des « - ». Les branches qui ajoutent une feature se nomment « ft-<featurename> ». Les branches qui résolvent des bugs « bg-<bugsolving> ».



Figure 1Exemple branches

### Issues

Chaque fonctionnalité/tâche fera l’objet d’une « Issue ». Lorsqu’un collaborateur travaille sur une issue, il se désigne comme « assignee ». Lorsque la tâche est terminée, il effectue une « pull request » de sa branche sur la branche de l’itération, en spécifiant l’issue concernée. De cette manière, la personne qui validera le merge fermera également l’ issue. Les issues sont en français.

Les personnes assignées à l’issue posteront l’état de leur travail dans l’issue en cas de besoin.

### KanPan

Nous avons intégré un projet KanPan de GitHub. Chaque issue fera l’objet d’une étiquette automatiquement créée. Nous verrons l’avancement et la répartition des tâches d’une meilleure manière qu’uniquement par l’onglet « Issue ».

Nous y avons inséré 4 colonnes :

* **Todo :** Regroupe les issues qui n’ont pas été commencées.
* **In progress :** Regroupe les issues qui sont en cours de traitement
* **Waiting to be validated** Regroupe les “Pull request” en attente de validation.
* **Done** Regroupe les « issues » et les « Pull request » terminées

Il y aura un kanban par sprint.

### Commit

Les commits seront signés sinon ils ne seront pas acceptés par GitHub. Ils sont écrits en français.

**Entête** : Très court, max 50 caractères Résumé des modifications apportées.

**Message :** Max 4-5 lignes, explique plus en détail ce qui a été changé, éventuellement pourquoi, mais pas comment. Le comment est obtenu en lisant les modifications par rapport à l’état précédent.

### Intégration continue

Une GitHub action a été configurée afin de tenter de faire la packaging de l’application sur la dernière version de Windows. Elle est déclenchée à chaque commit/merge sur la branche main et sur le sprint1.

La commande lancée est « mvn -B package --file pom.xml ». De cette façon, les tests avec JUnit seront automatiquement effectués.

# Description du projet

Le but de ce projet est d’implémenter un générateur de sites statiques. Le site sera généré à partir à partir de deux fichiers.

Un fichier JSON, YAML ou TOML. Ce fichier permettra de créer les métadonnées de la page dans le <head> ainsi que certaines configurations générales du site.

Un fichier Markdown. Ce fichier générera le contenu de la page dans le <body> ainsi que certaines métadonnées propres à la page.

L’utilisateur aura 4 commandes à disposition :

* **statique init** : Initialisera le répertoire donné en argument avec les fichiers pour créer le site.
* **statique clean** : Nettoie le répertoire de build.
* **statique serve**: Ouvre la page index.html sur le navigateur
* **statique build** : Génère le site en fonction des fichiers, le tout dans un dossier build.

# Sprint 1

## Réunion de début de sprint

Au début, nous ne comprenions pas le travail que nous devions faire. Nous nous sommes documentés sur les générateurs de sites statiques, notamment Hugo et Jekyll. Ainsi, nous avons pu nous diriger vers une première ébauche de ce que nous devions réaliser.

Nous avons divisé les stories en tâche que nous nous sommes attribuées. Comme les stories sont « simples », chaque story correspond à une tâche.

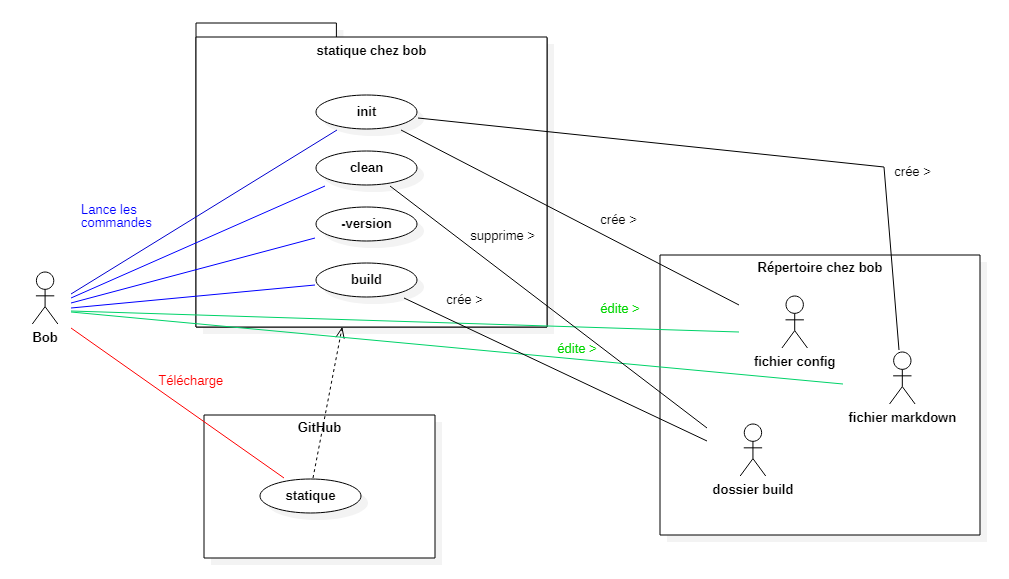
## Conception

### User requirement

Bob désire écrire et créer un site statique. Il peut le configurer globalement dans le fichier config et créer les pages grâce aux fichiers Markdown. Pour effectuer ce processus, il utilise les commandes init, clean et build.

### Use cases

Ce diagramme montre les utilisations possibles de l’application pour le sprint1.



## Collaboration

L’aspect collaboratif de ce sprint a été expliqué dans la section « Workflow » de la page 5. Dans l’ensemble, toutes les pratiques décrites ont été respectées sauf la nomenclature des branches qui a été mal respectée par moment.

## Stories

### Saisie des données structurées

En tant qu’utilisateur, j’aimerais que toutes les données structurées du site (p. ex. configuration, métadonnées, etc.) puissent être saisies dans un format uniforme tel que JSON, YAML ou TOML.

### Saisie du contenu

En tant qu’utilisateur, j’aimerais pouvoir saisir du contenu dans un format de markup simplifié tel que Markdown ou Asciidoctor qui sera compilé en HTML.

Nous prendrons en charge au minimum les annotations suivantes pour le contenu.  
 - # pour les titres 1  
 - ## pour les titres 2  
 - Du texte brut pour les paragraphes  
 - ![Une image](./un/chemin) pour les images

Format des pages  
En tant qu’utilisateur, j’aimerais pouvoir créer des pages contenant des métadonnées et du contenu.

Nous prendrons en charge les annotations suivantes pour les métadonnées.  
 - titre : <un titre> pour le titre de la page.  
 - auteur : <un nom> pour l’auteur de la page.  
 - date : <AAAA-MM-JJ> pour la date de la page.

Le contenu sera le même que la section « Saisie de contenu »

Les métadonnées seront délimitées par « --- ».

Affichage de la version  
En tant qu’utilisateur, j’aimerais exécuter la commande suivante de manière à afficher la version du générateur de site statique dans le terminal. « *$ statique –version »*

Initialiser un site statique  
En tant qu’utilisateur, j’aimerais exécuter la commande suivante de manière à initialiser un site statique. « *$ statique init /mon/site »*

Compiler un site statique  
En tant qu’utilisateur, j’aimerais exécuter la commande suivante de manière à compiler le site statique. *« $ statique build /mon/site ».*

Nettoyer un site statique  
En tant qu’utilisateur, j’aimerais exécuter la commande statique clean /mon/site de manière à nettoyer le site statique. En d’autres termes, la sous-commande clean doit supprimer le dossier /mon/site/build.

Amélioration de l’intégration continue  
Amélioration de l’intégration continue. En tant que développeur, j’aimerais que les pull requests ne puissent être mergées que lorsque la compilation et les tests ne produisent pas d’erreur.

Release et documentation  
En tant qu’utilisateur, j’aimerai qu’une release du générateur de site statique soit téléchargeable depuis l’onglet release de GitHub. Un tag git (v0.0.1) correspondant à la release doit être publié. La release doit également être accompagnée d’instructions d’installation reproductible.

### Tâches

#### Faire un choix de format entre JSON, YAML et TOML pour la saisie des données structurées [au mieux 3h, au pire 6h, attendu 4h]

La tâche a été réalisée à 2 (Alec et Quentin) et nous avons globalement passé 3h sur cette dernière en tenant compte des recherches, des petits prototypes et du rapport sur le choix de la technologie choisie. Nous pensions mettre plus de temps, mais nous avons vite trouvé une librairie qui convenait totalement à nos besoins.

#### Implémenter l’argument -version : [au mieux 1h, au pire 2h, attendu 1h30]

Implémenter la prise en charge de l’argument –version à la commande statique.

La tâche a été réalisée en 1h30. Tout s’est passé comme prévu. Nous nous attendions à perdre du temps dû à l’utilisation des paramètres de Picocli, mais nous avons réussi à les utiliser grâce à leur documentation en ligne. La version est directement lue depuis le fichier pom.

#### Configurer les GithubActions : [au mieux/au pire/ attendu : 5 min]

Temps total 5 minutes. La tâche a été faite à l’aide d’une simple GitHub actions.

#### Initialiser un site statique : [au mieux 1h, au pire 2h, attendu 1h30]

Implémenter la sous commande init. Elle prend un argument un chemin. Si le chemin n’existe pas, elle le créer. Elle ajoute au répertoire un fichier config.yaml et un fichier index.md.

La tâche a été réalisée en 1h40. L’utilisation des paramètres avec Picoli a pris plus de temps que prévu. De plus nous avons implémenté la possibilité de ne pas donner de chemin et travailler dans le répertoire courant. Ainsi le confort d’utilisation est augmenté.

#### Implémentation de la commande clean [au mieux 1h30, au pire 4h, attendu 2h]

Implémenter la commande clean a pris un peu plus de temps possible, car les tests ont pris plus de temps à réaliser que prévu. Effectivement nous avons réalisé des tests sans utiliser les autres commandes du générateur statique. Nous avons également recherché comment faire des AssertThrows() pour de futurs besoins. La tâche a donc été réalisée en 2h30.

#### Implémentation de la commande build [au mieux 1h, au pire 3h, attendu 2h]

Implémenter la sous-commande build n’a pas posé trop de problèmes. Elle prend un argument optionnel en paramètre qui spécifie le chemin du site à build. Si le chemin n’est pas spécifié, la commande va s’effectuer dans le dossier courant.

Les tests ont cependant pris plus de temps. En effet, les tests passaient en local, mais pas sur GitHub action, ce qui m’a perturbé et compliqué la tâche pour retrouver l’erreur. À cause de ce problème, la tâche a été effectuée en 2h30.

#### Interprétation d’un fichier Markdown [au mieux 1h, au pire 4h, attendu 2h]

Pour réaliser cette tâche, nous avons utilisé la librairie commonsmark qui nous a fait gagner énormément de temps. Très simple d’utilisation avec plein d’extensions possibles, la tâche a pris 2h30 pour l’interprétation ainsi que la « compilation » en HTML.

#### Compiler l’interprétation Markdown en HTML [au mieux 1h, au pire 3h, attendu 1h30]

Voir [Interprétation d’un fichier Markdown [au mieux 1h, au pire 4h, attendu 2h]](#_Interprétation_d’un_fichier). Ici, il a été simplement utile de créer des fichiers HTML à partir de la fonction créée pour le point cité.

#### Publier le sprint 1 dans les release GitHub : [au mieux 10 min, au pire 45, attendu 20]

À la fin du sprint, ajouter la release avec un document readme.md qui décrit l’installation.

## Code reuse

Nous avons estimé qu’il n’était pas nécessaire de créer notre propre parser. Nous avons donc cherché ce qui existait déjà. Nous avons trouvé plusieurs technologies qui se vantaient de parser des fichiers markdown et de créer les fichiers HTML résultants. Au bout du compte, nous avons choisi la libraire « commonmark-java ». Elle nous permet de générer des fichiers HTML à partir de fichiers markdown tout en supportant le langage YAML pour les métadonnées. Avec cette librairie nous faisons d’une pierre deux coups. Nous avons fait un petit prototype pour vérifier son fonctionnement et elle convient aux exigences du sprint1.

## Choix de la technologie pour la saisie de contenu

Le fichier de contenu du site devra être réalisé à l’aide d’une technologie. Ce dernier sera ensuite utilisé afin de créer un fichier HTML pour la réalisation du site statique.

Les 2 principales technologies possibles étaient :

* Markdown
* Asciidoctor

L’aboutissement de nos recherches peut être résumé avec une liste des avantages et inconvénients de chacune des technologies.

|  |  |  |
| --- | --- | --- |
|  | Avantages | Inconvénients |
| Markdown | * Très populaire * Simple d’utilisation * Facile à lire en java * Solutions pour transformer un fichier en HTML * Étudier en cours | * Nous n’en avons pas trouvé. |
| Asciidoctor | * Simple d’utilisation | * Surtout utilisé en Python ou Ruby * Beaucoup moins populaire que Markdown |

Nous avons donc choisi d’utiliser la technologie Markdown pour notre projet.

## Choix de la technologie pour les fichiers de configuration

Le fichier de configuration du site sera utilisé pour structurer les données de ce dernier et également transformer un fichier Markdown(.md) en fichier HTML.

3 principales technologies s’offraient à nous :

* JSON
* YAML
* TOML

Nous avons fait des recherches sur ces 3 technologies afin de choisir celle qui conviendrait le mieux à la réalisation de notre projet. Pour ceci nous avons d’abord regardé comment était composé un fichier JSON, YAML, TOML. Puis nous avons fait des recherches sur la potentielle existence d’une implémentation d’un programme Java utilisant une de ces technologies et un fichier Markdown pour créer / compiler des fichiers HTML.

L’aboutissement de nos recherches peut être résumé avec une liste des avantages et inconvénients de chacune des technologies.

|  |  |  |
| --- | --- | --- |
|  | Avantages | Inconvénients |
| JSON | * Très populaire * Simple d’utilisation * Technologies existantes pour parser un fichier en Java * Étudier en cours | * Pénible à écrire à la main |
| YAML | * Simple d’utilisation * Simple à écrire à la main * Technologies existantes pour parser un fichier en Java * Libraire pour interpréter le YAML et le Markdown | * Moins populaire que JSON |
| TOML | * Simple à écrire à la main | * Surtout conçu pour d’autres langages que Java * Moins populaire que JSON |

La technologie YAML sera donc notre choix malgré la popularité de JSON. Le point qui a fait pencher la balance en sa faveur est dû au fait que nous avons trouvé une libraire qui interprète le Markdown et le YAML comme on le souhaite pour le projet. Il est également plus agréable d’écrire un fichier YAML à la main qu’un fichier JSON.

## Test-first programming

C’est une pratique à laquelle nous ne sommes pas familiers. Nous n’avons pas réussi à la suivre de façon stricte. Nous avons imaginé les tests avant de coder l’implémentation des fonctionnalités. Les tests unitaires ont souvent été implémentés en même temps que la fonctionnalité. Comme nous avons gagné en expérience, c’est une méthodologie que nous allons tenter de mieux utiliser au cours des futurs sprints.

## Commit early, commit often

La plupart des tâches étaient relativement courtes. En général, cette pratique a été suivie inconsciemment. Cependant l’implémentation de la commande Build était un peu plus conséquente que les autres. Absorbés par l’idée de faire fonctionner notre code avant de le commit, la pratique « commit often » a été mise de côté.

Dans l’ensemble cela a bien fonctionné. Nous allons quand même plus nous concentrer sur cette méthode au cours des sprints avenirs. Pour y remédier, nous allons commit à la fin de chaque heure de programmation.

## Réunion de fin de sprint

Nous avons discuté de ce qui a bien fonctionné et pas fonctionné.

|  |  |
| --- | --- |
| Bien | Mal |
| Issues | Test-first programming |
| Nommage des commits | Commit often |
| Répartition du travail |  |
| Implémentation des fonctionnalités |  |

Dans l’ensemble la planification, du temps de travail, était correcte. Cependant, lors de tâches techniques « inconnues », le temps prévu rejoignait souvent le temps « au pire ».

La structure du code est convenable. Différentes classes ont été créées pour les différentes fonctionnalités. Le code a été pensé pour être évolutif.

Toutes les fonctionnalités demandées ont été implémentées. Malheureusement, nous n’avons pas compris l’utilisation du fichier config. De ce fait, nous l’avons laissé de côté en attendant d’y voir plus clair lors des prochaines itérations.

Finalement, nous sommes contents du travail réalisé et comptons continuer dans cette lancée.

# Sprint 2

## Réunion de début de sprint

Comme d’habitude la réunion de début de sprint était un peu chaotique. Comment débuter ? Vers quoi se diriger ? Qu’est-ce qui est le mieux ?

Le noyau de ce sprint est le moteur de template. La première étape consiste à définir le bon moteur à utiliser. Pour notre projet, le moteur idéal est :

* Utilisable en java
* Capable de fusionner des templates (menu.html -> layout.html)
* Capable d’insérer plusieurs éléments (titre du site, titre de la page et contenu)

Nous avons ensuite subdivisé les stories en tâche. Dans l’ensemble une story égale une tâche sauf le moteur de template qui était plus conséquent.

Au terme de cette réunion, le travail à réaliser se dessine plus nettement. Que le spectacle commence !

## Choix du moteur de template

Pour choisir le moteur de template que nous allions utiliser pour le projet, nous avons d’abord fait des recherches pour savoir ce qui existait.

Nous avons fait ressortir la liste suivante des solutions existantes et intéressantes :

* Handlebars
* Java Server Pages (JSP)
* Thymeleaf
* FreeMarker
* Groovy
* Jade4j
* Chunk Templates

Nous avons commencé explorer en vitesse chacune de ces solutions.

Nous avons ensuite fait des recherches plus profondes en commençant par Handlebar (conseillé par le professeur pour le projet / le cours). Cette solution contenant exactement tout ce qui était nécessaire à la réalisation du sprint, nous avons décidé de l’utiliser.

Voici les autres moteurs de templates qui auraient également pu être utilisés facilement selon nous :

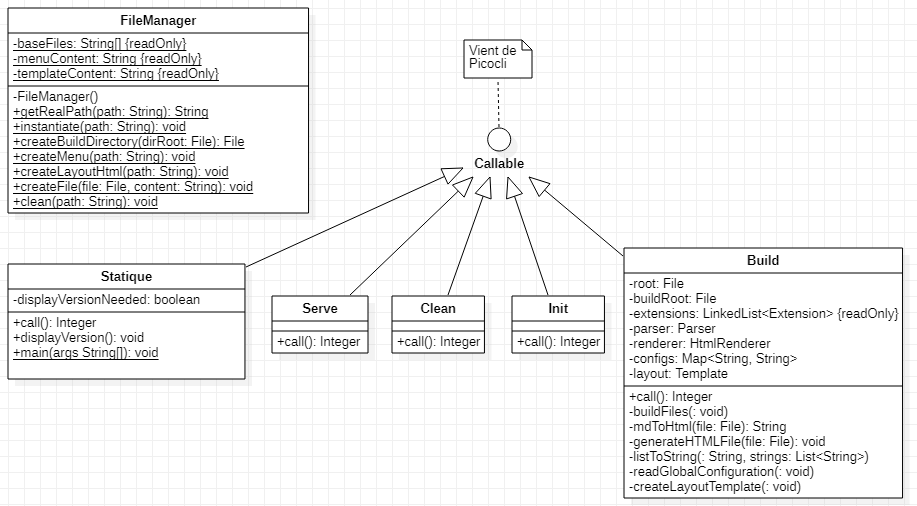
* Thymeleaf
* FreeMarker
* Chunk Templates

Voici les autres solutions qui n’étaient pas si simples à utiliser, n’avait pas les notations que l’on recherchait ou qui demandaient beaucoup plus d’investissement dans leur compréhension selon nous :

* Java Server Pages (JSP)
* Groovy
* Jade4

## Conception incrémentale et décomposition

### Diagramme de classe



### Diagramme de séquence

Une image contenant table

Description générée automatiquement

Le principe est simple, lors d’une génération, on récupère d’abord la configuration générale du site et on compile le template afin de pouvoir utiliser ces informations dans tous les fichiers suivants.

Ensuite, on crée les fichiers HTML à partir de tous les fichiers makdown, en y ajoutant les informations globales du site.

## Stories

### Moteur de template

*« En tant qu’utilisateur, j’aimerais pouvoir utiliser un moteur de template (p.ex. Handlebars1) de manière à enrichir mes pages HTML avec des éléments d’interfaces (header, footer, menu, etc.). En d’autres termes, j’aimerai que le HTML correspondant à mes fichiers Markdown soit injecté dans des pages HTML plus complètes à l’aide d’un template. »*

**Résultat :** La commande build récupère toutes les informations du site et les fichiers markdowns pour les générer les fichiers HTML en fonction du template prédéfini. Pour ce faire, nous avons utilisé Handlebars qui subvenait à nos besoins facilement.

À noter qu’il est nécessaire de changer les informations du menu.html à la main, et que les liens ont un risque d’être faux s’ils sont dynamiques. Par exemple, lorsqu’un fichier qui inclut menu.html est dans un sous-dossier.

### Serveur HTTP

*« En tant qu’utilisateur, j’aimerais exécuter la commande suivante de manière à pouvoir visualiser le résultat de la compilation du site Internet dans un navigateur Web.*

*$ statique serve /mon/sites »*

**Résultat** : La commande serve doit indiquer le dossier du projet (celui généré par init). Elle recherche le fichier index.html dans /build. Si le fichier n’existe pas ou que le fichier ne peut être ouvert, une explication est affichée dans la console.

### Continuous delivery

*« En tant que développeur, j’aimerais que la création et la publication d’une nouvelle release soit automatisée à l’aide de GitHub actions. En d’autres termes, j’aimerai que le fichier jar ou le fichier zip de la release soit publié de manière automatique lorsqu’un tag correspondant à une version (p. ex. v0.0.2) est publié sur git. »*

**Résultat** : Pour ce faire nous avons réutilisé la GitHub action create-release[[1]](#footnote-1) Elle nous permet d’ajouter le fichier target/statique.zip en tant qu’asset ce qui est très pratique pour télécharger la partie utilisateur du projet.

En guise de bonus, nous avons réussi à lier le fichier README du répertoire en tant que description des release.

Si nécessaire dans un futur proche, il serait possible de remplacer le contenu du README par un changelog automatique[[2]](#footnote-2) avec les commits.

Ainsi en faisant *$git tag vx.x.x* et *$git push –tag*, une nouvelle release est créée.

## Tâches

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Mettre à jour la commande Init  Faire en sorte que la commande init crée le dossier template et les fichiers menu et template. | | | | |
| Temps | Au mieux : 1h | Au pire : 2h30 | Prévu : 2h | Réel : 1h30 |
| Commentaires : | Tout s’est correctement passé. Les tests ont pris un peu plus de temps que prévu. Une gestion maladroite des chemins les a rendus plus longs à développer. | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Implémenter la commande Serve  Faire en sorte que la commande serve ouvre le site dans le navigateur Web. | | | | |
| Temps | Au mieux : 45h | Au pire : 2h | Prévu : 1h | Réel : 1h30 |
| Commentaires : | J'ai perdu du temps sur la prévision. Lorsque je transformais un fichier en URI, cela ne fonctionnait pas à cause de "/./" dans le chemin. Au début je pensais que c'était à cause des espaces dans mon chemin. Du coup j'ai pris du retard le temps de comprendre d'où venait le problème. Désormais l’URI est parse et remplace les « ./ » par des strings vides. | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Modifier la commande Build  Cette tâche a été réalisée en pair programming par Quentin et Alec.  La commande build doit être modifiée de telle façon :   * Les fichiers HTML doivent maintenant se baser sur le fichier template/layout.html en utilisant un moteur de template * Il faut supprimer le dossier template copié dans le dossier build | | | | |
| Temps | Au mieux : 3h | Au pire : 6h | Prévu : 4h30 | Réel : 4h25 |
| Commentaires : | La réalisation de cette tâche s’est assez bien déroulée. Nous avons juste eu de la peine au début, car il fallait comprendre comment utiliser Handlebars.  Nous avons réussi à être dans les temps malgré notre mauvaise compréhension d’Handlebars, car nous avions très peu de tests à écrire. Effectivement, il suffisait de modifier ceux créés auparavant. | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Publication de release automatique avec GitHub Action Faire en sorte qu’une release soit automatiquement créée lorsqu’un tag est ajouté. | | | | |
| Temps | Au mieux : 45h | Au pire : 2h | Prévu : 45h | Réel : 2 |
| Commentaires : | Temps prévu 1h, au mieux 45min, au pire 2h, temps réel = 2h  J'ai effectué sur un repo personnel différents tests d'actions.  Je pensais arriver facilement à mes fins. Malheureusement, je n'arrivais pas à faire à upload le fichier target/statique.zip en tant qu'asset. J'ai testé diverses actions.  https://github.com/marketplace/actions/automatic-releases  https://github.com/softprops/action-gh-release  https://github.com/actions/create-release  Je souhaitais vraiment faire en sorte que l'utilisateur puisse directement télécharger le bon asset. J'ai finalement réussi avec la dernière.  J'ai également perdu du temps, car les GA ne se mettaient pas forcément à jour lorsque j'éditais le fichier yml correspondant. Je devais recréer un repository de test à chaque fois que je souhaitais tester une nouvelle méthode. | | | |

## Refactoring

Étape essentielle, mais délicate. Chacun de nous a travaillé sur des commandes spécifiques. Chaque commande interagit avec le système de fichier. Nous avons regroupé ces interactions dans une classe FileManager indépendante des classes de commandes. De cette façon, la gestion du système de fichier est centralisée et la logique des commandes est encapsulée dans des classes respectives à chacune.

L’outil de « refactor » d’IntelliJ a été utilisé des façons suivantes :

* Renommer nos noms de variables et la signature de nos classes afin de les rendre plus explicites. Le risque était de ne pas renommer des variables qui ont le même nom, mais qui ont des contextes différents.
* Extraire des fonctions là où la complexité méritait d’être encapsulée. Ici la difficulté était de mettre en commun ce qui était redondant (au sein d’une classe et entre les classes) afin d’éviter d’avoir plusieurs fonctions sensiblement similaires.

Dans l’ensemble le refactoring est positif , ça allège le code et encapsule la difficulté. Toutefois c’est une opération délicate. Il ne faudrait pas casser un système fonctionnel. Chaque opération de refactor a débouché sur l’exécution des tests pour vérifier que le système correspond toujours aux spécifications.

## Tests d’intégration et tests système

Dans la lancée du sprint 1, nous avons enrichi le projet de test pour couvrir les nouvelles fonctionnalités :

* L’application génère-t-elle les nouveaux fichiers ?   
  template/menu.html et template/layout.html
* L’application intègre-t-elle le contenu correctement.

Les éléments précédents ont fait l’objet de nouveaux tests et/ou de l’adaptation de ceux déjà en place.

Finalement quelques tests utilisateurs ont été réalisés

|  |  |
| --- | --- |
| Description | Validé ? |
| Je peux télécharger l’application au format zip, depuis l’onglet release de GitHub | Validé |
| Sur l’onglet release, il est expliqué comment utiliser l’application. | Validé |
| Si j’exécute $statique -version, cela affiche que l’application est en version v0.0.2 | Validé |
| Si j’exécute $statique init, l’application crée les fichiers[[3]](#footnote-3) dans le dossier courant. | Validé |
| Si j’exécute $statique init <un/chemin/>, l’application créer les fichiers dans le dossier en paramètre. | Validé |
| Si je modifie mon index, que je rajoute une page et que je l’ajoute au menu.html. La commande $statique build génère mon site correctement (Titre du site et des pages dans le head, le contenu est présent). | Validé |
| La commande $statique build <un/chemin/de/dossier> si le chemin indiqué contient un répertoire build, le site s’ouvre dans un navigateur. | Validé |

## Automatisation

À ce stade, le projet est déjà automatisé à plusieurs endroits :

**KanBan**Le Kanban ajoute automatiquement une étiquette lorsqu’une issue est créée. L’étiquette va dans la colonne « TODO ». Lorsque l’issue est assignée, l’étiquette passe dans « In progress ». Finalement lorsque l’issue est terminée, l’étiquette passe dans « Done ». Plus besoin de changer manuellement les états.

**Tests**À chaque pull/push sur les branches de sprint et la branche main. Les tests sont exécutés automatiquement et permettent d’avoir une vue en temps réel de ce qui fonctionne.

**Release automatique**Lors de l’ajout d’un nouveau tag qui commence par v, une nouvelle release est créée par GitHub action. Le dossier statique.zip est téléchargé comme asset et le readme du projet ajouté comme description de la release.

## Commit early, commit often

Cette pratique a été suivie de manière inconsciente. Lorsqu’une fonction ou deux étaient créées, nous effectuions un commit. Évidemment, c’est quelque chose qui a été fait inconsciemment. De manière générale, nous avons tendance à commit lorsque notre morceau de code fonctionne. Ainsi l’aspect Early est parfois remplacé par « when it works ». Toutefois, la promesse du sprint1 « Commit à la fin de chaque heure » a globalement été tenue.

Cependant, chose que nous devons travailler : push often. Il est arrivé que certains d’entre nous ne push pas leur travail et que d’autres doivent attendre, car leur tâche repose sur le travail du précédent.

## Réunion de fin de sprint

Arrivé à la fin de ce sprint, les exigences sont tenues. Le générateur est fonctionnel. Il fait l’objet de tests automatiques ainsi que de tests utilisateurs.

L’architecture de l’application est contrôlée. Nous avons factorisé le code et encapsulé la complexité. De cette façon, nous osons espérer partir sur de bonnes bases pour le sprint 3.

Nous avons une bonne gestion de la répartition des tâches et de leur enchaînement. Nous avons su éviter les conflits lors des pull requests.

L’application est, certes, fonctionnelle, mais elle pourrait être améliorée. Actuellement l’utilisateur doit mettre à jour manuellement le fichier menu.html. Lors du build, il est réaliste d’imaginer une génération automatique des liens lorsque des fichiers markdown sont trouvés.

# Sprint 3

## Réunion de début de sprint

C’est la dernière ligne droite. Après une rapide analyse, nous constatons que ce sprint porte plus sur l’ajout d’outil qui aide au pilotage du projet plutôt que l’ajout de nouvelles fonctionnalités. Toutefois, il y en a quand même une, implémenter la génération du site statique à la volée.

Nous avons séparé le travail comme suit :

|  |  |  |
| --- | --- | --- |
| Berney Alec | Forestier Quentin | Herzig Melvyn |
| * Manuel utilisateur * Code coverage | * Génération du site statique à la volée | * Javadoc * Code Quality * Code benchmarking |

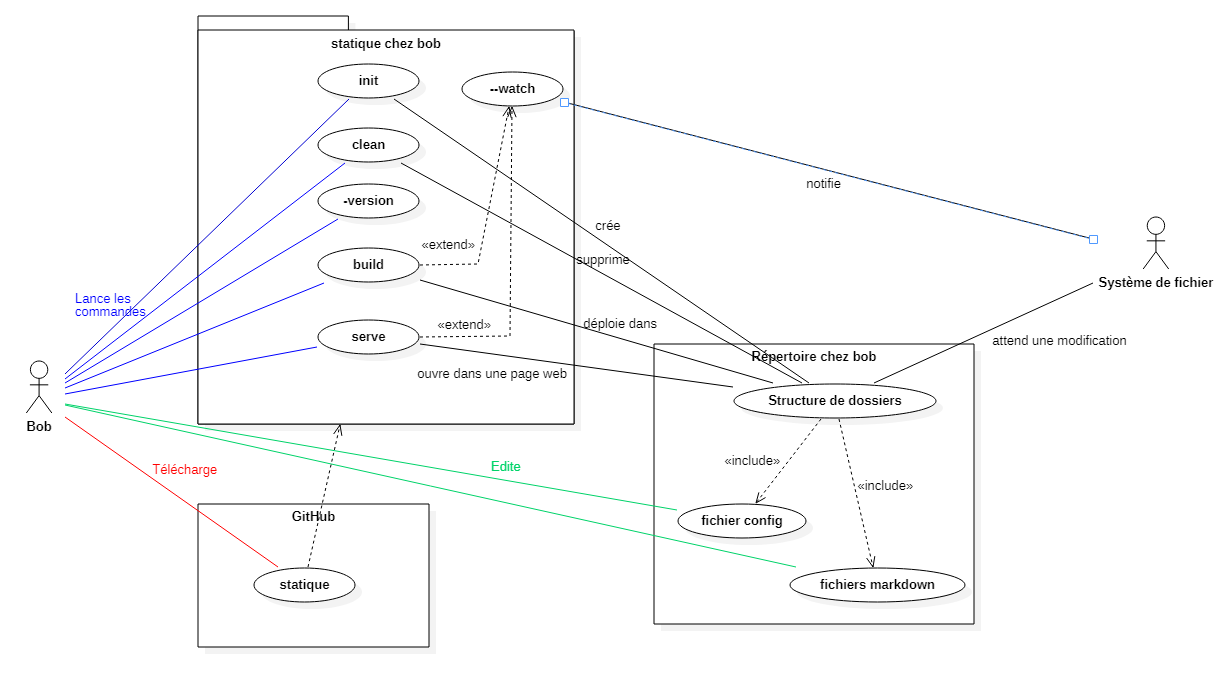
Comme ce dernier sprint est « plus court » que les autres à cause du Jeudi de l’ascension, nous avons séparé les tâches le plus équitablement possible en fonction du niveau d’expertise de chacun.

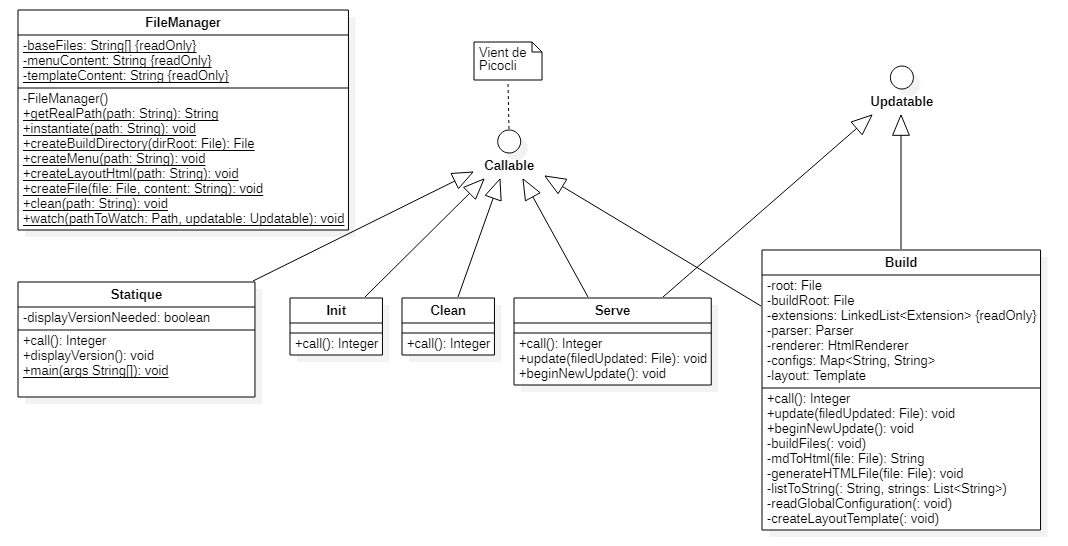
Berney Alec a principalement travaillé sur les tests de l’application. Il est donc le plus à même d’intégrer le calcul de leur couverture. De plus, il a souvent manipulé le générateur dans son ensemble. Cela en fait, le candidat idéal pour la réalisation de la documentation utilisateur.

Forestier Quentin a été le principal développeur de l’algorithme de production du site. De ce fait, nous lui avons attribué la tâche d’intégrer la génération à la volée.

Quant à Herzig Melvyn, il a été la personne qui a le plus travaillé sur les outils d’intégration continue, sur Github avec les Github actions ou avec Maven. Ces trois tâches correspondent le plus à ses capacités.

## Conception incrémentale et refactoring





On peut voir que l’impact n’est pas démesuré. Il suffit d’avoir une méthode détectant les modifications sur un chemin donné, et de mettre à jour les Updatable inscrits en attente.

Il suffit de s’inscrire à cette méthode que lorsque l’option --watch est spécifiée, et le tour est joué.

Le code existant n’a pas besoin d’être modifié, il ne reste qu’à ajouter l’implémentation de l’interface Updatable afin que les mises à jour effectuées soient conformes à nos besoins.

## Stories

### -Génération du site statique à la volée

*« En tant qu’utilisateur, j’aimerai que les commandes build et serve puissent régénérer le site statique à la volée, c’est-à-dire de manière automatique lorsque des changements sont effectués sur le système de fichier. Pour cela, ajoutez une option --watch aux commandes build et serve et assurez-vous que leur exécution ne se termine pas à la fin du premier build. »*

**Résultat :** Les 2 commandes ont l’option –watch implémentées. Build va regénérer le site entièrement lorsqu’une modification est détectée **dans le dossier racine du site**. Serve va se réexécuter lorsqu’une modification **dans le dossier build** est effectuée, autrement dit, lorsque la commande build est exécutée.

**Question ouverte :** « Faut-il recompiler tous les fichiers lors de chaque changement ? »

Il est certain que recompiler tous les fichiers est coûteux. Il serait donc préférable de recompiler seulement le fichier modifié.

Cependant, quelques problèmes surviennent. Si l’utilisateur modifie un des fichiers du dossier template (layout.html ou menu.html) ou config.yaml, il faudrait alors recompiler tous les fichiers.

De plus, l’objet permettant de détecter les modifications en Java peut envoyer plusieurs événements pour l’ajout d’un seul caractère. Le fichier sera donc recompilé plusieurs fois, pour la même modification.

Le code deviendrait rapidement très complexe à mettre en place, et à faire évoluer par la suite, vu le nombre de conditions et cas à mettre en place. C’est pourquoi nous avons décidé de regénérer le site en entier, et pour ne pas rebuild à chaque modification, nous avons implémenté un système permettant de ne le faire qu’une seule fois lorsque tous les événements ont été envoyés.

### Manuel utilisateur

*« Rédigez un manuel utilisateur reproductible qui guide l’utilisateur lors de la création d’un site statique avec votre générateur. »*

**Résultat** : le manuel utilisateur est disponible sur le github sous le nom « userManual.md ». Il est également disponible dans la release v 0.0.3.

### Javadoc

*« Configurez Maven de manière à pouvoir publier une javadoc au format HTML. Ajoutez la Javadoc à votre release »*

**Résultat :** La génération de la documentation a été faite au moyen du plugin   
maven-javadoc-plugin qui génère la javadoc au format HTML. À l’aide de maven-assembly-plugin, nous zippons la javadoc. Finalement, nous avons modifié la Github action release.yml, pour charger la javadoc automatiquement en temps qu’asset de la release.

### Code coverage

*« Configurez Jacoco ou un outil équivalent pour calculer le degré de couverture de votre code par des tests unitaires et d’intégration. Ajoutez les données de couverture à votre rapport. »*

**Résultat :** le résultat du code coverage et ces données sont ci-dessous.

Nous avons utilisé Jacoco pour générer ces données.

Ces données n’étant pas disponibles sur le repository github pour des raisons de confidentialité, il est nécessaire d’effectuer la commande « mvn clean install » à la racine du repository cloné.

Elles sont ensuite consultables dans le dossier « target\site\jacoco » et en affichant le fichier « index.html » dans le navigateur.

Il est possible de recréer ces données à tout moment en exécutant la commande « mvn clean install » dans le répertoire principal.

Voici le rendu global :

Une image contenant texte

Description générée automatiquement

Premièrement, on remarque que nous ne couvrons pas notre code à 100%. Cela s’explique, car différents points ne peuvent pas être testés. Nous parlons ici notamment de la commande « serve » et diverses autres classes. Ces points seront détaillés plus tard dans le rapport de code coverage.

### Package gen.command

Si on sélectionne le package « gen.command » en clique sur le « + », on affiche les résultats suivants :

Une image contenant table

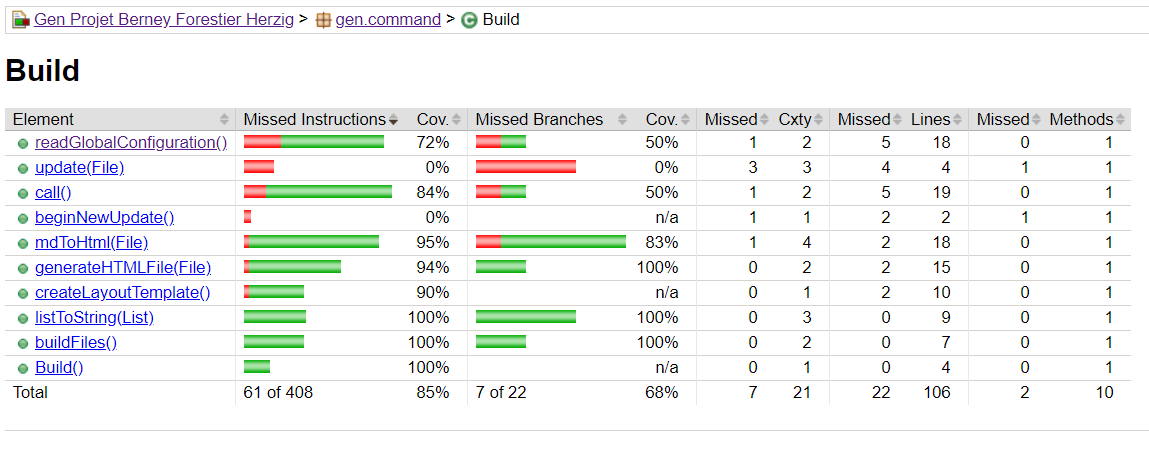
Description générée automatiquement

On remarque que la commande serve n’a pas du tout été testée, car cette commande est impossible à test à l’aide de tests automatiques selon nous. Elle est cependant testée dans les tests d’intégration et système.

Il est possible de sélectionner chaque classe afin d’afficher son code coverage par fonction. Pour chacune des méthodes de chaque classe, il également possible de consulter directement le code coverage dans le code pour voir qu’elles instructions ne sont pas testées.

Nous allons afficher le code coverage de chaque classe du package « command » sauf « Serve ».

**Classe Build**



En parcourant chacun des fichiers, on remarque que les parties non testées sont soit des méthodes privées, soit des « catch » d’exceptions liés à la lecture de fichiers. Ces erreurs de lecture de fichiers étant parfois dures à simuler, il est normal de ne pas pouvoir les tester automatiquement.

**Classe Clean**

Une image contenant texte

Description générée automatiquement

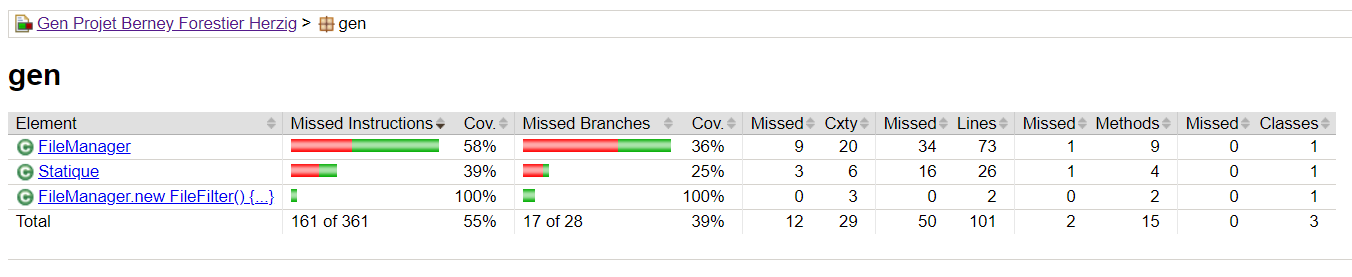
**Classe Init**

Une image contenant table

Description générée automatiquement

### Package gen

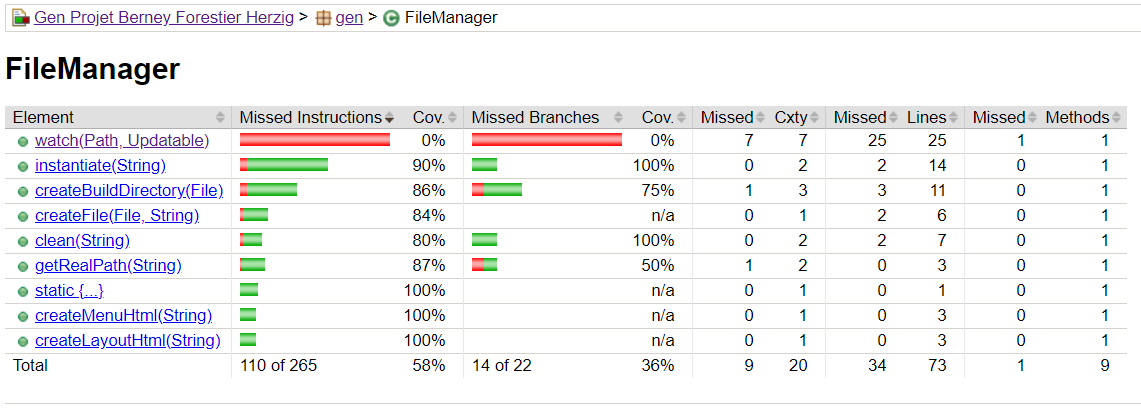
Passons maintenant au reste du code dans le package « gen » et ceci en cliquant sur le « + ».



On remarque que les classes « FileManager » et « Statique » n’ont pas un code coverage de 100% et cela s’explique par un nombre fonctions privées non testées. Nous allons y revenir plus tard dans le rapport du code coverage.

Comme expliqué dans le précédent package, nous allons de nouveau parcourir chaque classe.

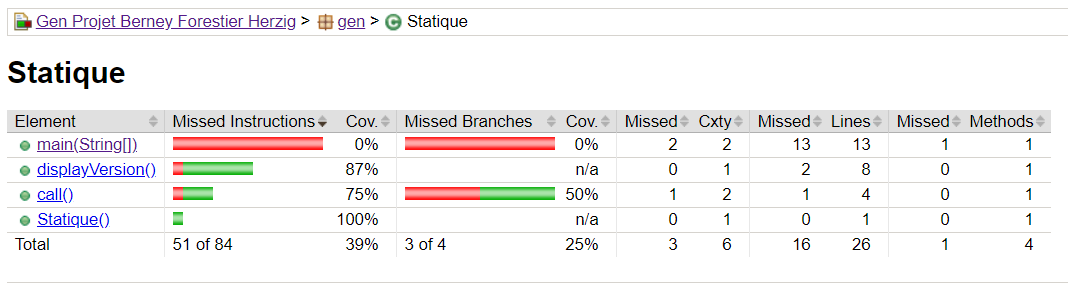
**Classe FileManager**



On remarque que la méthode « watch » ne fait l’objet d’aucun test automatique et cela est normal, car cette méthode demande l’intervention de l’humain afin de modifier le fichier markdown et les fichiers de configuration. Cette partie nous paraissait compliquée à mettre en place et nous prendrait beaucoup de temps. De plus, cette partie est testée dans les tests d’intégrations et système et n’est donc pas mise de côté.

Le reste est simplement des exceptions non levée et difficile à réaliser automatiquement.

**Classe Statique**



On remarque que la méthode « main » ne fait l’objet d’aucun test automatique et cela est normal, car cette méthode est utilisée afin de lancer le programme lors d’une utilisation classique.

Le reste est simplement des exceptions non levée et difficile à réaliser automatiquement.

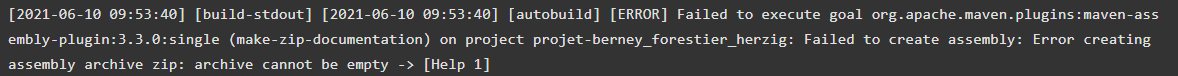
### Code quality

*« Configurez LGTM , SonarQube ou un outil équivalent pour détecter des bugs et des vulnérabilités dans votre code. Expliquez en quoi l’un de ces outils vous aura permis d’améliorer vos pratiques de développement. »*

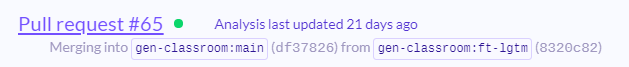
**Résultat :** Grâce à l’intégration continue, les résultats sont directement ajoutés sur le readme du répertoire.



Le répertoire est testé à chaque pull request sur le main. Malheureusement depuis l’ajout de la javadoc, la phase d’extraction de lgtm échoue.



Nous avons tenté de réparer le problème en ajoutant un dossier « apidocs » non vide, mais ça n’a pas résolu le problème. Par manque de temps, nous n’avons pas réussi à réparer le bug. Heureusement nous avons réussi à faire fonctionner lgtm avant le merge de la javadoc.

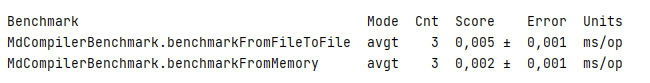


Toutefois ce commit ne prend pas en compte l’implémentation de la génération à la volée.

### Code benchmarking

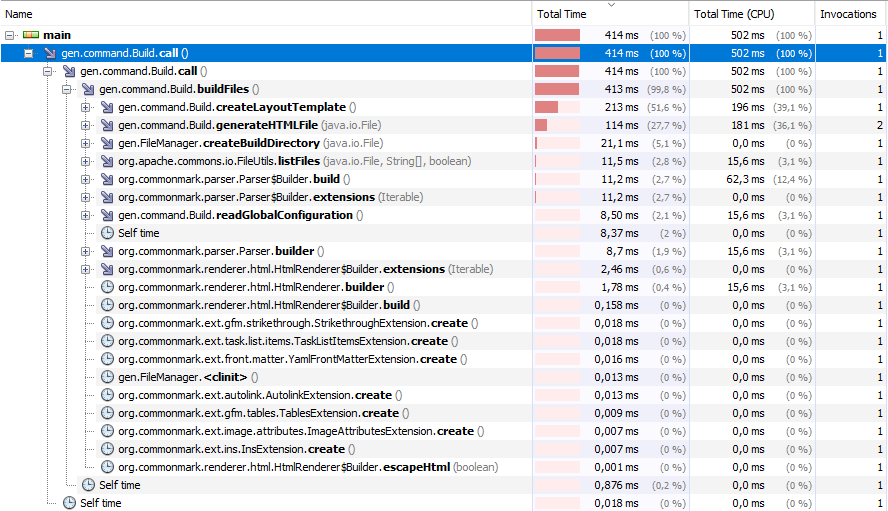
*« Utilisez JMH pour mesurer les performances de votre compilateur markdown ou de votre moteur de rendu. Utilisez VisualVM pour profiler une exécution de la commande build. Incluez ces données dans votre rapport et discutez d’une éventuelle optimisation (sans nécessairement l’implémenter). »*

**Résultat :** (Pour ne pas polluer la release, la branche *ft-benchmark* n’a pas été merge avec le reste du projet.)

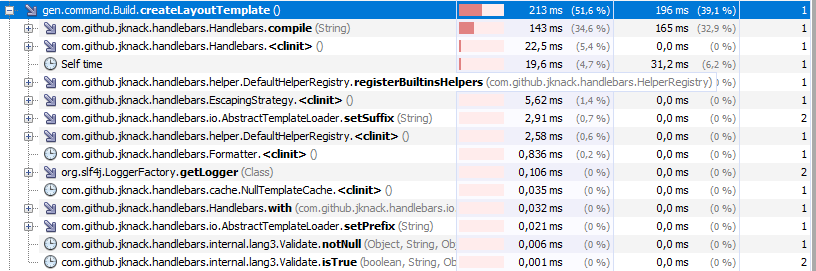
Deux benchmarks ont été mis en place: l’un, convertit un texte markdown depuis la mémoire et l’ autre effectue la même opération depuis et vers un fichier. La chaîne traitée faisait la taille raisonnable de 5Ko.

Comme le montre les résultats précédents, la lecture et l’écriture dans les fichiers prennent plus de la moitié du temps de l’opération. Le fichier n’étant pas petit et la compilation étant courte, nous en déduisons que le compilateur markdown est performant.

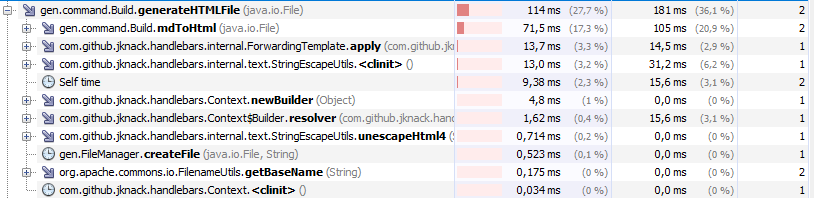
Voici les performances de la commande build :



Nous remarquons que l’opération la plus longue est la fonction buildFiles. 75% de son temps d’exécution est séparé dans deux sous-appels à createLayoutTemplate et generateHTMLFile.

Profil de createLayoutTemplate :

Ici, le temps d’exécution est le résultat des méthodes obligatoire d’handlebars. Nous ne pouvons pas vraiment améliorer ses performances.

Profil de generateHTMLFile :

Dans ce cas nous voyons que la fonction la plus longue est mdToHtml. Elle est chargée de lire chaque fichier md du site, de lire le contenu, le passer à handlebar et de l’écrire dans un fichier HTML. Ici encore les opérations sont standards à nos librairies.

Optimiser les performances dépend peu de notre volonté. La manière la plus optimale serait de tester d’autres librairies et de toutes les profiler pour choisir la meilleure. Malheureusement, nous n’avons pas le luxe de nous offrir autant de temps.

## Tâches

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Génération du site statique à la volée  Implémentation de l’option --watch dans les commandes Build et Serve. | | | | |
| Temps | Au mieux : 4h | Au pire : 10h | Prévu : 9h | Réel : 8h |
| Commentaires : | 2 approches ont été testées. La recompilation de tous les fichiers, ou alors la compilation du fichier modifié. La première méthode est retenue, par sa simplicité et son évolutivité. | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Manuel utilisateur  Rédigez un manuel utilisateur reproductible qui guide l’utilisateur lors de la création d’un site statique avec votre générateur. L’ajouter à la release et le rendre accessible sur le github. | | | | |
| Temps | Au mieux : 1h30 | Au pire : 4h | Prévu : 2h30min | Réel : 2h |
| Commentaires : | Melvyn ayant déjà réalisé une partie du manuel utilisateur lors du premier sprint, il suffisait de l’adapter et ajouter les nouvelles fonctionnalités ainsi que de le tester. J’ai également pu effectuer les tests d’intégration et systèmes en même temps. | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Générer une documentation Javadoc  Tout au long du développement de l’application, nous avons commenté notre code au format java doc. Cette tâche consiste générer la documentation du code et l’ajouter à la release. | | | | |
| Temps | Au mieux : 30 min | Au pire : 1 h | Prévu : 45 min | Réel : 35 min |
| Commentaires : | La génération de la documentation a été faite au moyen du plugin  maven-javadoc-plugin qui génère la javadoc au format HTML.  Penser à initialiser la variable JAVA\_HOME avec le répertoire d'installation de java.  Pour ce faire, aller dans: Settings > Build, Execution, Deployment > Build Tools > Maven > Runner Dans mon cas j'ai ajouté: JAVA\_HOME=C:\Program Files\AdoptOpenJDK\jdk-11.0.8.10-hotspot | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Calculer la couverture du code  Configurez Jacoco5 ou un outil équivalent pour calculer le degré de couverture de votre code par des tests unitaires et d’intégration. Ajoutez les données de couverture à votre rapport. | | | | |
| Temps | Au mieux : 1h | Au pire : 3h | Prévu : 1h30min | Réel : 1h45 |
| Commentaires : | Utiliser Jacoco était assez simple, il suffisait d’ajouter la dépendance dans le fichier « pom.xml ». J’ai cependant fait des recherches pour afficher ce rapport de code coverage dans IntellJ et écris un petit tutoriel disponible dans le README du git. J’ai également passé un certain temps à essayé d’intégrer un tag de code coverage sur le README dans un projet forké, mais sans succès. | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Calculer la qualité du code  Détecter les bugs et les vulnérabilités du code | | | | |
| Temps | Au mieux : 1 h | Au pire : 2 h | Prévu 1 h 30 | Réel : 4 h |
| Commentaires : | J'ai commencé à tenter d'intégrer LGTM. Je n'ai pas réussi à l'utiliser immédiatement. Nos fichiers sources étaient ignorés, car considérés comme du code auto généré. J'ai donc cherché à utiliser le fichier de configuration lgtm.yml, en vain... De plus, chaque test prenait environ 5 minutes à s'exécuter. J'ai donc décidé d'approcher sonarQube. Toutefois cette solution demande une plus grande installation et les options d'intégration sont moins évidentes. De ce fait, je me suis repenché sur LGTM, j'ai créé plusieurs répertoires privés afin de tester différentes variantes du fichier de configuration. J’ai également installé l’application LGTM directement sur le répertoire afin d’effectuer une vérification du code à chaque pull request sur main. Pour avoir accès au panneau de contrôle en ligne, contacter [melvyn.herzig@heig-vd.ch](mailto:melvyn.herzig@heig-vd.ch). | | | |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Calculer un benchmarking du code.  Cette tâche consiste à effectuer un benchmark de notre compilateur markdown avec JMH et un profilage de la commande build avec visualVM. | | | | |
| Temps | Au mieux : 1 h | Au pire : 3 h | Prévu : 2 h | Réel : 2 h |
| Commentaires : | La prise en main de JHM a été rapide grâce à la démonstration du Prof. Chapuis.  Penser à initialiser la variable JAVA\_HOME avec le répertoire d'installation de java.  Pour ce faire, aller dans: Settings > Build, Execution, Deployment > Build Tools > Maven > Runner Dans mon cas j'ai ajouté: JAVA\_HOME=C:\Program Files\AdoptOpenJDK\jdk-11.0.8.10-hotspot  À l’aide de maven-assembly-plugin, nous zippons la javadoc. Finalement, nous avons modifié la Github action release.yml, pour charger la javadoc automatiquement en temps qu’asset de la release. | | | |

## Refactoring

Au cours du sprint nous n’avons pas fait d’opération de refactor notable. Le gros du travail a été effectué durant le sprint2.

Afin d’éviter les conflits lors du refactoring au sein d’un groupe, nous avons utilisé « code with me » de intelliJ. Nous pouvions tous travailler simultanément sur une instance d’intelliJ afin que chaque modification soit immédiatement propagée chez les collègues.

Ainsi, aucun risque de conflit, car il n’y a qu’un seul commit.

## Tests d’intégration et tests système

Depuis le sprint 2, aucun test unitaire / automatisé n’a été ajouté, car le seul ajout était compliqué à tester automatiquement. L’ajout en question est l’argument « watch ».

Finalement quelques tests utilisateurs ont été ajoutés et réalisés :

|  |  |
| --- | --- |
| Description | Validé ? |
| Je peux télécharger l’application au format zip, depuis l’onglet release de GitHub | Validé |
| Sur l’onglet release, il est expliqué comment utiliser l’application. | Validé |
| Si j’exécute $statique, cela affiche la liste des commandes utilisables | Validé |
| Si j’exécute $statique <commande inexistante>, cela affiche le message d’erreur, puis la liste des commandes utilisables. | Validé |
| Si j’exécute $statique -version, cela affiche que l’application est en version v0.0.3 | Validé |
| Si j’exécute $statique init, l’application crée les fichiers[[4]](#footnote-4) dans le dossier courant. | Validé |
| Si j’exécute $statique init <un/chemin/>, l’application créer les fichiers dans le dossier en paramètre. S’il n’existe pas, créer également le dossier. | Validé |
| Si je modifie mon index, que je rajoute une page et que je l’ajoute au menu.html. La commande $statique build <un/chemin/de/dossier> génère mon site correctement (Titre du site et des pages dans le head, le contenu est présent). | Validé |
| Si j’exécute $statique build <un/chemin/de/dossier> et que le chemin indiqué contient au moins un répertoire « template » et les fichiers « config.yaml » et « index.md », génère le site correctement. | Validé |
| Si j’exécute $statique build <un/chemin/de/dossier> et que le chemin indiqué ne contient pas le répertoire « template » et /ou les fichiers « config.yaml » et « index.md », affiche un message d’erreur. | Validé |
| Si j’exécute $statique --watch build <un/chemin/de/dossier> et que le chemin indiqué contient au moins un répertoire « template » et les fichiers « config.yaml » et « index.md », génère le site correctement. Si je modifie un fichier, régénère le site correctement. | Validé |
| Si j’exécute $statique serve en ayant créé le site à la racine, affiche le site dans un navigateur. | Validé |
| Si j’exécute $statique serve <un/chemin/de/dossier> en ayant indiqué le bon dossier, affiche le site dans un navigateur. | Validé |
| Si j’exécute $statique serve en ayant pas créé le site à la racine, mais dans un autre dossier, affiche un message d’erreur. | Validé |
| Si j’exécute $statique serve <un/chemin/de/dossier> en ayant indiqué le mauvais dossier, affiche un message d’erreur. | Validé |
| Si j’exécute $statique --watch serve <dossier> en ayant indiqué le bon dossier, affiche le site dans un navigateur. Le site se réaffiche dans le navigateur après chaque modification. | Validé |

## Automatisation

Comme cité plus haut dans les stories, nous avons automatisé les points suivants :

**Javadoc:** La documentation est générée puis zippée automatiquement par maven grâce aux plugins Javadoc et assembly. Nous avons automatisé son téléchargement dans les assets de la release grâce au workflow release.yml

**Code quality:** L’analyse de la qualité du code a été automatisée. Pour ce faire, nous avons ajouté l’application de lgtm à notre répertoire. De cette façon, à chaque pull request sur la branche main, une analyse est effectuée. Un résumé du code coverage est directement disponible dans le readme grâce à deux snippets.

## Commit early, commit oftern

Ce sprint était différent des précédents, car il incluait beaucoup de tâches sans commit à proprement parler. La seule story qui nécessitait des commit était l’implémentation de la génération à la volée.

Même cette dernière était relativement courte : une fonction, une interface et l’appel de cette fonction dans deux classes.

Ainsi nous avons difficilement pu mettre en place la partie commit « often ».

## Réunion de fin de sprint

Le sprint touche à sa fin, dans l’ensemble les stories ont pu être implémentées, mais le travail n’est pas parfait.

Dans une itération future, il faut réussir à réparer lgtm et également améliorer la génération automatique sans avoir besoin de regénérer l’ensemble du dossier.

# Bilan

Nous sommes arrivés au terme de ce petit projet de générateur de site statique. Nous avons pu pratiquer différents aspects du génie logiciel :

* Modélisation
* Planification
* Réaction face aux problèmes
* Anticipations des problèmes
* Collaborations de groupe
* …

L’ensemble du projet a pu être implémenté conformément aux stories sauf lors du dernier sprint ou nous n’avons pas atteint notre niveau d’exigence.

Dans l’ensemble, nous sommes satisfaits du travail que nous avons fourni, de manière constante, durant ce semestre. Nous reconnaissons que tout n’est pas parfait, mais nous vous garantissons avoir fait du mieux que nous le puissions avec les conditions semestrielles.

Nous avons été amenés à sortir de notre zone de confort et c’est avec des capacités plus riches que nous terminons ce projet, car nous sommes devenus plus souples dans nos méthodes de travail.

1. <https://github.com/actions/create-release> [↑](#footnote-ref-1)
2. <https://github.com/marketplace/actions/release-drafter> [↑](#footnote-ref-2)
3. *Index.html, config.yaml, template/menu.html, template/layout.html* [↑](#footnote-ref-3)
4. *Index.html, config.yaml, template/menu.html, template/layout.html* [↑](#footnote-ref-4)