Berney Alec, Forestier Quentin, Herzig Melvyn

GEN – 22.04.2021

Portfolio

GÉnie LOgiciel



Table des matières

[Introduction 3](#_Toc66818081)

[Outils 3](#_Toc66818082)

[Questions préalables 3](#_Toc66818083)

[Pourquoi signer les commits avec GPG 3](#_Toc66818084)

[Commit ealry, commit often 3](#_Toc66818085)

[Que doit contenir un message de commit ? 3](#_Toc66818086)

[Conventions de codage 4](#_Toc66818087)

[Langues 4](#_Toc66818088)

[Accolades 4](#_Toc66818089)

[Entête de fichier 4](#_Toc66818090)

[Commentaires 4](#_Toc66818091)

[Classes 4](#_Toc66818092)

[Méthodes 4](#_Toc66818093)

[Nommage 4](#_Toc66818094)

[Workflow 5](#_Toc66818095)

[Processus piloté ou agile 5](#_Toc66818096)

[Collaboration 5](#_Toc66818097)

[Branches 5](#_Toc66818098)

[Issues 5](#_Toc66818099)

[KanPan 6](#_Toc66818100)

[Commit 6](#_Toc66818101)

[Intégration continue 6](#_Toc66818102)

[Description du projet 7](#_Toc66818103)

[Sprint 1 7](#_Toc66818104)

[User requirement 7](#_Toc66818105)

[Architecture 8](#_Toc66818106)

[User stories 8](#_Toc66818107)

[Saisie des données structurées 8](#_Toc66818108)

[Saisie du contenu 8](#_Toc66818109)

[Format des pages 8](#_Toc66818110)

[Affichage de la version 8](#_Toc66818111)

[Initialiser un site statique 9](#_Toc66818112)

[Compiler un site statique 9](#_Toc66818113)

[Nettoyer un site statique 9](#_Toc66818114)

[Saisie des données structurées 9](#_Toc66818115)

[Saisie des données structurées 9](#_Toc66818116)

[Release et documentation 9](#_Toc66818117)

# 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 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 et 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**: ?
* **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.

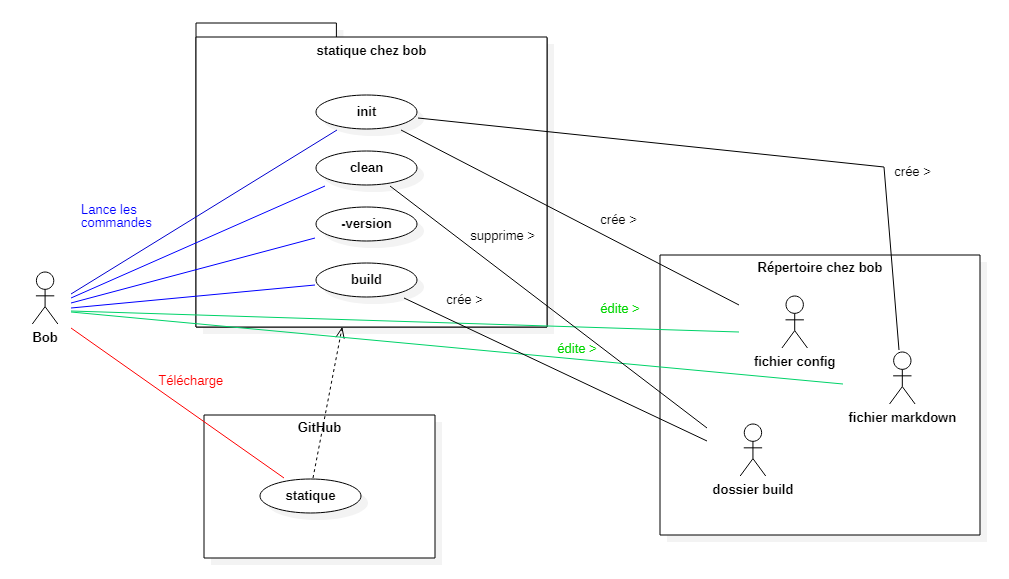
## Conception

### 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’aimerai 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]

A 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.

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

# Sprint 2

## Réunion de début de sprint

<Description>

## Choix du moteur de template

<commentaire>

## Conception incrémentale et décomposition

Une image contenant table

Description générée automatiquement

<commentaires>

## Stories

<Stories>

### 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 long à développer. | | | |

## Refactoring

<commentaires>

## Tests d’intégration et tests systèmes

<commentaires>

## Automatisation

<commentaire>

## Commit early, commit often

<commentaire>