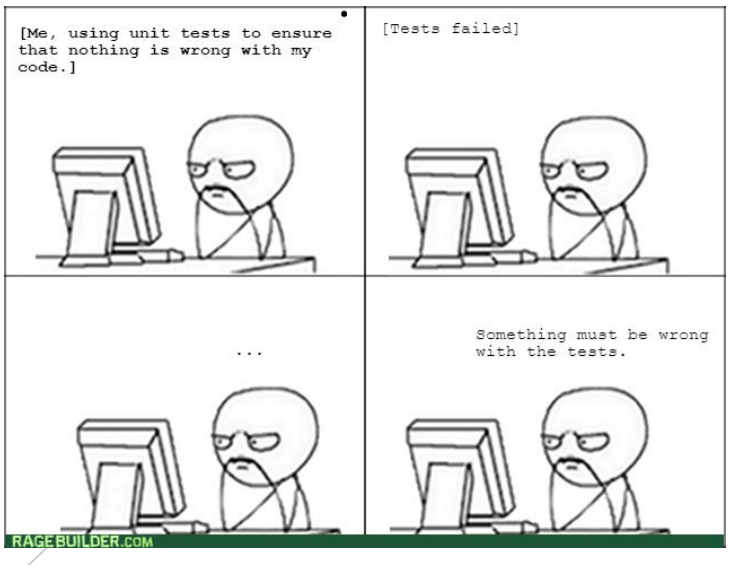
11/06/2021

PROJET GEN

Générateur de site statique



PRSV

Table des matières

1 Introduction

Il était une fois dans le magnifique royaume de l'HEIG, quatre jeunes étudiants souhaitant faire leurs preuves dans ce monde sans pitié qu'est l'informatique. Ils se réunirent pour former l'équipe PRSV et mener à bien un ambitieux projet proposé par le responsable du village. Celui-ci consistait à réaliser un générateur de site statique.

Le chemin vers la victoire était semé d'embûches et consistait en trois dangereuses étapes nommées Sprint, chacune ayant ses propres pièges et dangers. Parmi les principaux défis à relever, il y avait

- Réaliser 4 commandes : init, build, serve et clean
- Implémenter un moteur de template avec Handlebar, l'indomptable roi des Templates
- Mettre en place une Watch Api, reine des modifications en temps réel
- Se soumettre au sévère jugement de L'Hydre formée des 4 têtes : Jacoco, JMH, VisualVM9 et SonarQube

La légende racontait que quiconque parvenait à remplir les 3 sprints obtenait le St-Graal.

Ce portfolio présente les processus et choix effectués par cette courageuse équipe d'aventurier. Les techniques d'organisation les plus innovantes et modernes vous seront ainsi présentées.

Lire ce rapport, c'est avant tout débuter un magnifique voyage au sein de la constellation infinie de la gestion de projet. Vous en ressortirez avec des étoiles plein les yeux et un regard confiant sur vos prochains projets à mener (tout comme nos aventuriers le furent).

1.1 Choix du processus(piloté/agile)

En débutant le projet, nous ne connaissions pas la nature du projet ni comment faire un site statique. Il était donc difficile de piloter ce projet. Par conséquent, nous avions choisis le processus agile.

Le processus piloté est intéressant si on maitrise bien les technologies utilisées et qu'on connait le cahier des charges. Par exemple, Ryan pour son travail de fin de CFC avait utilisé un processus piloté avec un diagramme de Gantt très détaillé pour faire son site web car il maîtrisait déjà bien le PHP et pouvait estimer correctement le temps nécessaire aux diverses tâches.

2 Collaboration

2.1 Répartition des tâches

Au sein du kanban, nous nous sommes réparti les différents tâches présentes dans le backlog. La création d'une "issue" liée à la tâche était ensuite réalisée ainsi qu'une "Pull request".

Le kanban était surtout utilisé en début de sprint. Seuls trois colonnes se sont finalement révélées nécessaires : To do(future), In progress et Done.

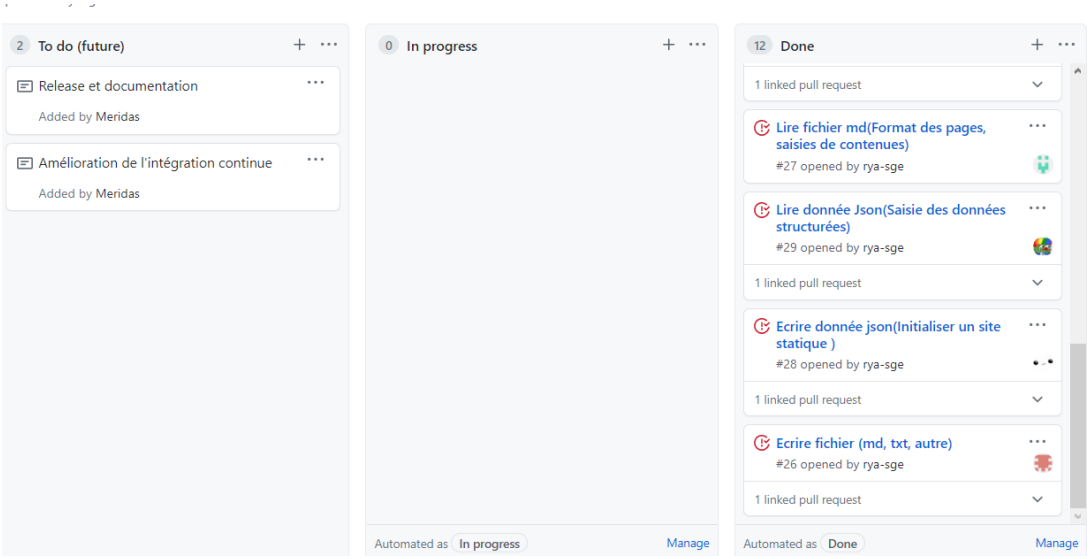


Figure 1 : état du Kanban le 01.04

2.2 Présentation

- Workflow

Nous avons appliqué le principe du workflow :

1. Création des issues dans le Kanban (éventuellement directement sur le git)
2. Attribution de l'issue
3. Création d'une branche dédiée.
4. Commit régulièrement
5. Pull Request

Une fois la tâche terminée, la personne générait une pull request. Celle-ci était alors vérifiée par un autre membre du groupe et mergée si tout était bon. Sinon, des modifications étaient demandées ou une nouvelle issue était créée. La review consommait certes le temps d'un autre membre du groupe, mais elle permettait de s'assurer du bon fonctionnement du code pour ne pas avoir (ou éviter au maximum) à revenir dessus plus tard, en assurant la qualité de ce qui serait poussé vers notre repo. Lorsque des modifications étaient demandées, les nouveaux commits venait alors directement sur la "pull request".

- Commit
    - Nous avons appliqué la méthode "commit early, commit often"
    - Les commits des grosses release doivent être signés. Nous avons décidé de ne de pas signer les petits commits par simplification, parce que nous avons vite vu que c'était contraignant.
- Règlement des désaccords

En cas de désaccords sur la façon de s'organiser ou réaliser le projet, nous utilisons le site Plouf qui permet d'effectuer un tirage au sort. Nous l'avons par exemple utilisé pour déterminer si les rapports se faisaient sur Google Doc ou One Drive. Plouf a choisi pour nous One Drive, ce qui nous a ainsi évité une perte de temps considérable sur un choix technologique n'ayant que peu d'incidence sur le projet. Lien du site : https://plouf-plouf.fr/

2.3 Moyen de communication

2.3.1 Telegram

Nous avons utilisé Telgram pour fixer les RDV et organiser les réunions.

2.3.2 Discord

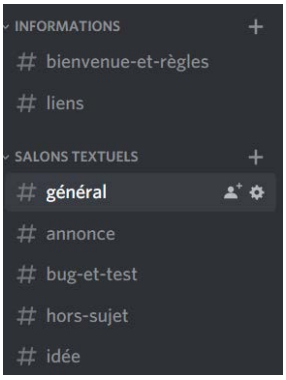


Figure 2 : représentation du serveur le 01.04

Nous avons utilisé Discord comme moyen de communication, grâce à un serveur que nous avons créé. Différents salons textuels ont été créés : liens, général, bug-et-test, hors-sujet, idée et des salons vocaux.

Discord permet, contrairement à des groupes de messagerie classique, de séparer les différents sujets traités en différents salons, ce qui réduit le "bruit" des conversations. De plus, nous avons fait plusieurs salons vocaux permettant à des membres de s'isoler entre eux s'ils devaient collaborer sur des fonctions qui étaient appelées par d'autres classes.

Discord propose également d'autres fonctionnalités intéressantes, par exemple si un membre souhaite savoir si les autres membres sont disponibles, il peut utiliser les fonctions de "broadcast" implémentée par Discord à travers le "@here" pour attirer l'attention de chacun.

3 Stories - Liste des tâches

La liste des taches présentées ici est un résumé. Certaines tâches peuvent par conséquent être manquantes.

3.1 Sprint 0

- Création classe main
- Création d'un projet Maven
- Ajout d'une licence
- Créer et commencer la rédaction d'un README
- Création des commandes : build, serve, clean et new

3.2 Sprint 1

3.2.1 - Résumé des tâches

Pour les tâches, nous les avons réparties comme suit :

Tâches principales

- Cmd clean : nettoyer le site statique
- Cmd build : compiler un site statique
- Cmd init: (+ création du fichier json)

Sous-tâches

- Lire fichier json(configuration)
    - Saisie des données structurées
- Ecrire fichier json(configuration)
- Lecture de fichiers md
- Ecriture des fichiers md

- Transformer markdown -> html
    - Format des pages
    - Utilisé par la commande build
- Générer release (avec assurance d'installation facile)
- Affichage de la version
- Release et documentation (rapport)
- Amélioration de l'intégration continue

3.2.2 Temps de travail

On a malheureusement oublié d'estimer le temps de chaque tâche.

Pour le nombre d'heures travaillées, nous nous sommes vu le 1er dimanche du sprint où nous avons chacun travaillé environ 6 heures, d'abord tout le groupe puis par équipe de 2. Nous avions décomposé les 1ères tâches en 2 groupes :

1) Écrire et lire le JSON
2) Lire et écrire des fichiers md + transformer md -> html.

Ensuite, nous avons avancés chaque vendredi durant les périodes de laboratoires sur le projet puis on s'est encore vu jeudi 1er avril pendant 3 heures avant de faire le rapport.

3.3 Sprint 2

Avec estimation du temps en minutes. Il y a une différence de +-40% par rapport au temps attendu pour les versions pessimistes et optimistes

Tâches	Description	T opt	T estimé	T pess	T réalisé
Création d'un default template	Template d'exemple "layout.html", qui pourrait être modifié plus tard	18	30	42	45
Init: création du layout, menu	Création de layout et menu avec la commande init	36	60	84	45
Build - Utilisation du moteur	Selon le template, écrire le contenu dans le fichier html	54	90	126	1380
handler - classe utils	Faire une classe utils pour le moteur handlebar	72	120	168	120
Continous delivery	Automatiser la publication de releases	36	60	84	180
Serveur http	"visualiser le résultat de la compilation du site Internet dans un navigateur Web."	90	150	210	350
Somme		**306**	**510**	**714**	**2120**

Comme on peut le voir dans le tableau, nous avions bien trop sous-estimé la charge de travail des tâches en général.

En particulier la tâche en rapport avec Handlebar qui nous a donné beaucoup de difficultés et a coûté les trois semaines de travail pour deux membres du groupe. La partie 4.2 de ce document explique plus en détails ces difficultés

D’autres tâches se sont rajoutées par la suite et ne sont pas listées ci-dessus. Ce sont notamment des améliorations faites après la réception du retour du premier sprint, par exemple l’écriture du README.

3.3.1 Moteur

Lors de la génération du fichier page.html (commande build), le générateur de site statique utilise le moteur pour

- injecter la propriété titre du fichier config.json
- injecter la propriété titre du fichier page.md
- inclure le fichier menu.html dans la page,
- injecter le HTML produit par la compilation du fichier markdown page.md

3.3.2 Commande Serve

Pour la commande serve, nous avons décidé qu’elle servira uniquement à lancer le serveur HTTP dans le dossier build, mais n’exécutera pas la commande Build en même temps.

Néanmoins, si le temps le permet, nous pourrions ajouter une option qui exécute le build en même temps.

La commande serve doit être suivie par le chemin du dossier racine.

Pour s’exécuter, serve appelle la classe Server qui utilise la classe staticHandler. Le code a été inspiré par cette source : https://github.com/ianopolous/simple-http-server.

3.3.3 Issues restantes

- Créer une image valide avec init

La commande init créée un fichier png. Ce fichier png n'est pas une image valide. Cette issue avait pour objectif de créer une image valide. Par manque de temps, cette issue n'a pas été réalisée et

3.4 Sprint 3

3.4.1 Tableau

Tâches	Description	T opt	T estimé	T pess	T réalisé
WatchAPI : observer les changements		240	400	560	150
WatchApi : compiler si changements		150	250	350	150
Build option --watch		60	100	140	180
Serve option --watch	Ajouter la watch api à la commande serve	60	100	140	60
Test serve option --watch	Si réalisable	120	200	280	60
Test build option --watch	Si réalisable	60	100	140	60
Code benchmarking: JMH8	Utilisez JMH8 pour mesurer les performances de votre compilateur markdown ou de votre moteur de rendu.	54	90	126	60
Code benchmarking: VisualVM9	Utilisez VisualVM9 pour profiler une exécution de la commande build	54	90	126	25
Code benchmarking: VisualVM9 -rapport	Incluez ces données dans votre rapport et discutez d'une éventuelle optimisation (sans nécessairement l'implémenter).	54	90	126	30
Ajouter la javadoc dans les release	Ajoutez la Javadoc à votre release. Cela peut se faire automatiquement grâce à Maven	72	120	168	160
Configurer Javadoc	Configurez Maven de manière à pouvoir publier une javadoc au format HTML4.	54	90	126	120
Code quality	Configurez LGTM6 , SonarQube7 ou un outil équivalent pour détecter des bugs et des vulnérabilités dans votre code.	90	150	210	120
Code coverage	Configurez Jacoco5 pour calculer le degré de couverture de votre code par des tests unitaires et d'intégration. Ajoutez les données de couverture à votre rapport.	90	150	210	60
Mettre à jour les diagrammes existants	Ajouter les composants créés pour la tâche "option --watch" => PAS fait	60	100	140	60

Diagramme d'activité	Notamment pour le server --> PAS fait	180	300	420	120
Test manuel du système	Tester le manuel pour savoir si il est compréhensible	36	60	84	90
Manuel utilisateur	Rédiger un manuel utilisateur	60	100	140	120
Somme		1494	2490	3486	1625
Somme en heures		24.9 h	41.5 h	58.1 h	27.1 h
Temps estimé par personne		6.3 h	10.4 h	14.6 h	6.8 h

3.4.2 Explication

Comme vous pouvez le constater, cette fois nous avons majoritairement surestimé le temps que chaque étape prendrait. Dans le Sprint 2, nous avions sous-estimé certaines étapes, notamment celle de Handlebar, qui avait pris 30 fois plus de temps environ que prévu. Cette fois, nous avons calculé "plus large" afin de nous laisser du temps, mais nous constatons que dans l'ensemble, nos temps optimistes étaient plus proches de la réalité (à part quelques exceptions).

Ainsi, la somme des temps passés est inférieure à la somme des temps optimistes. Nous constatons donc que, une "mauvaise expérience" durant le Sprint2 nous a rendu pessimiste quant au temps à accorder à des technologies inconnues (par exemple la Watch API, ou le code BenchMarkindiag). Ainsi, s'il devait y avoir un Sprint4 nous prendrions certainement le "juste milieu" entre l'optimisme du Sprint2 et le pessimisme du Sprint3 en matière de planification.

3.4.3 Génération site statique à la volée --watch

L'option --watch a été ajouté pour les commandes build et serve.

Pour la commande serve, l'ajout de l'option watch va avoir pour conséquence la création d'un nouveau thread qui lance la commande build avec l'option watch. Ainsi le site sera build à chaque nouvelle modification par ce thread. Une fois le server arrêté par l'utilisateur avec exit, alors le thread est lui aussi arrêté avec un appel à interrupt.

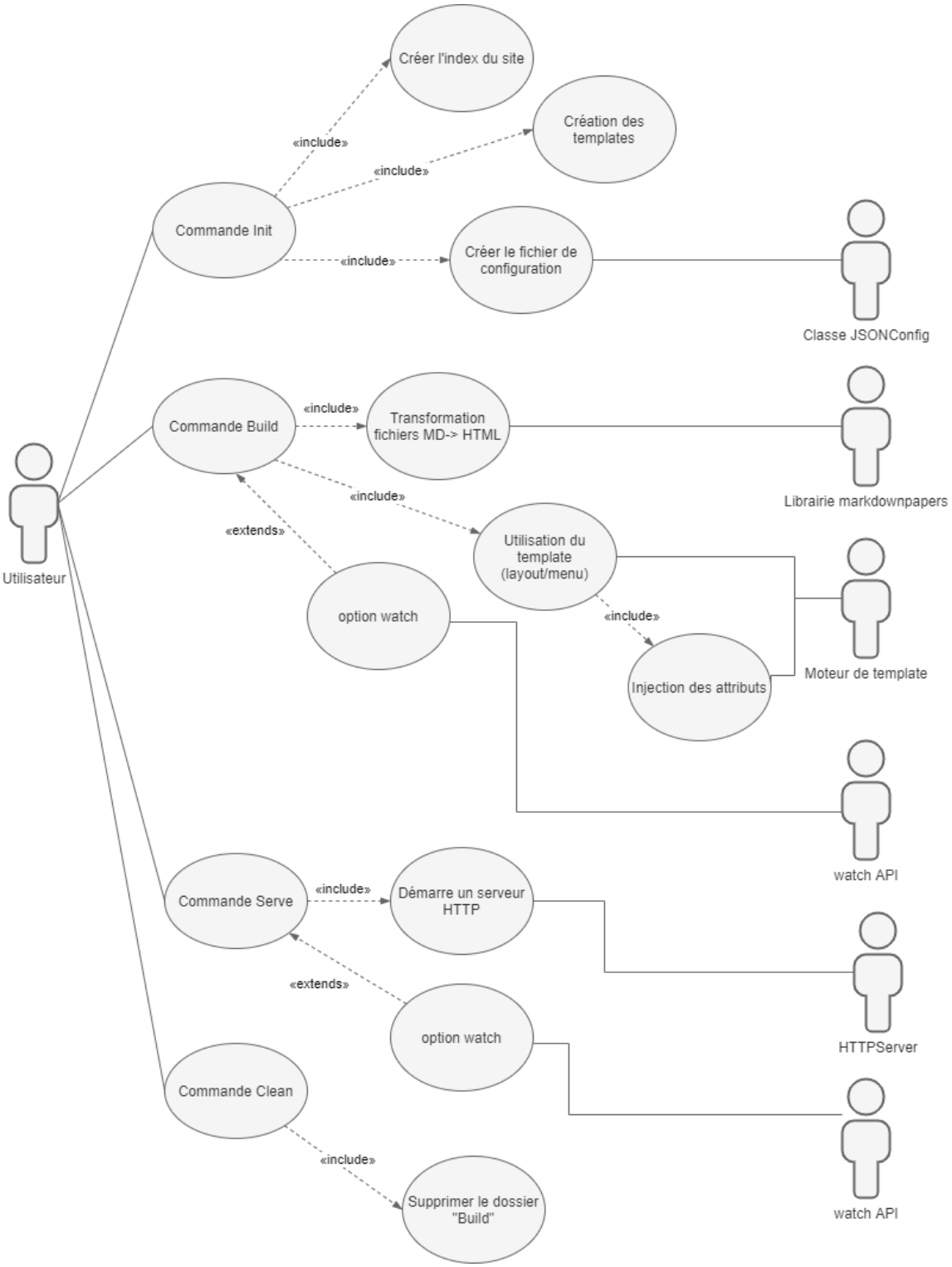
A chaque modification d'un fichier, le site est entièrement build à nouveau. On pourrait améliorer ça pour build uniquement le fichier concerné afin d'avoir des meilleures performances.

Sources :

- https://howtodoinjava.com/java8/java-8-watchservice-api-tutorial/
- https://docs.oracle.com/javase/tutorial/essential/io/notification.html

4 Conception

4.1 Cas d'utilisation



Ce schéma a été réalisé sur: https://www.diagrams.net/

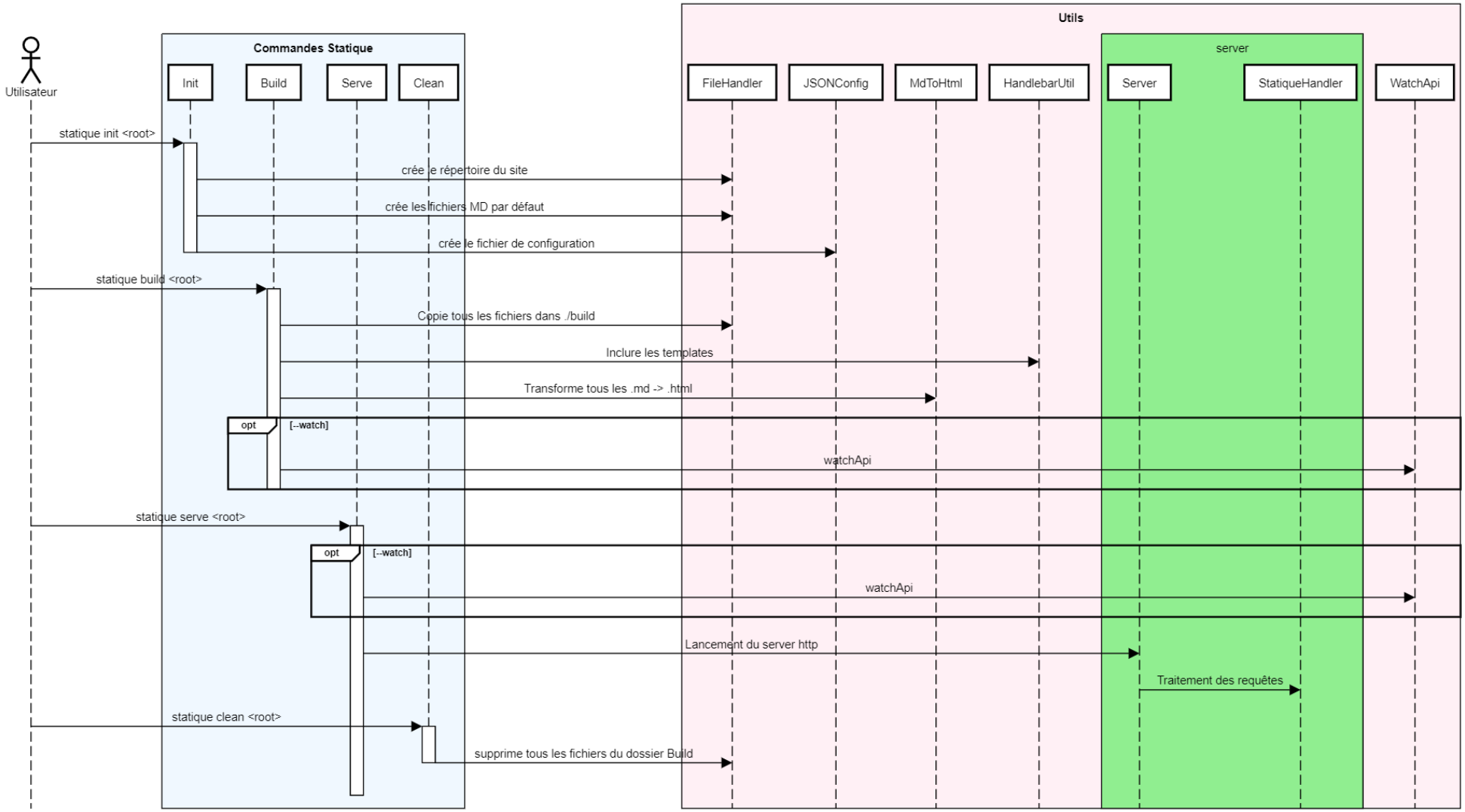
4.2 Séquence

Ce diagramme de séquence montre de manière succincte et sans entrer dans les détails le fonctionnement du site statique dans sa globalité. Dans le sprint 3, il est prévu de réaliser des diagrammes plus précis pour les différentes fonctionnalités.

Le diagramme a été créé avec l'outil en ligne* : https://sequencediagram.org/

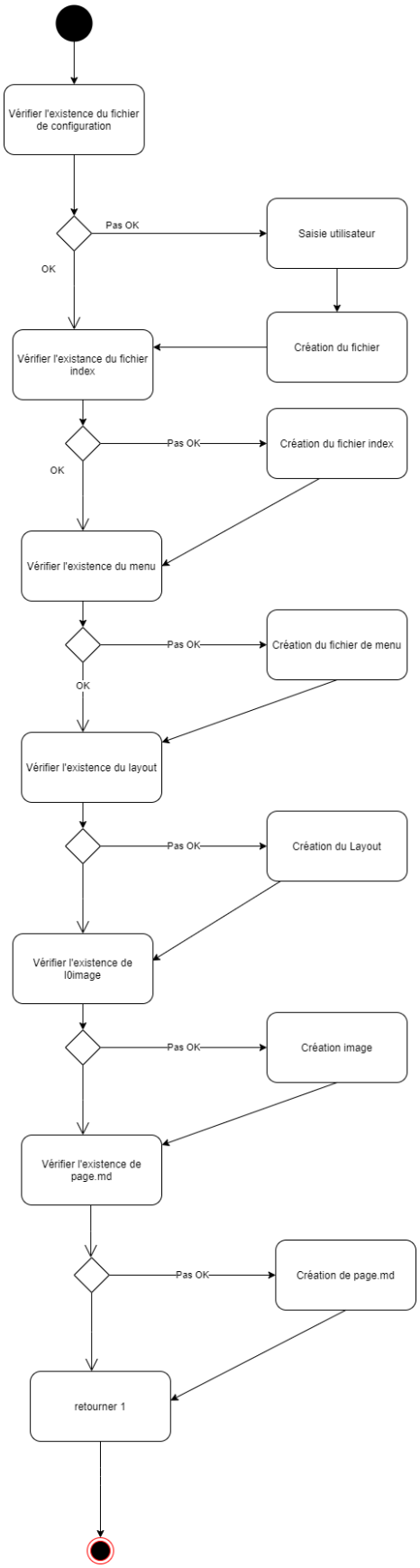
* excellent outil au passage, qui permet de créer des diagrammes de séquences plus ou moins complexes très facilement

Création d'un site statique

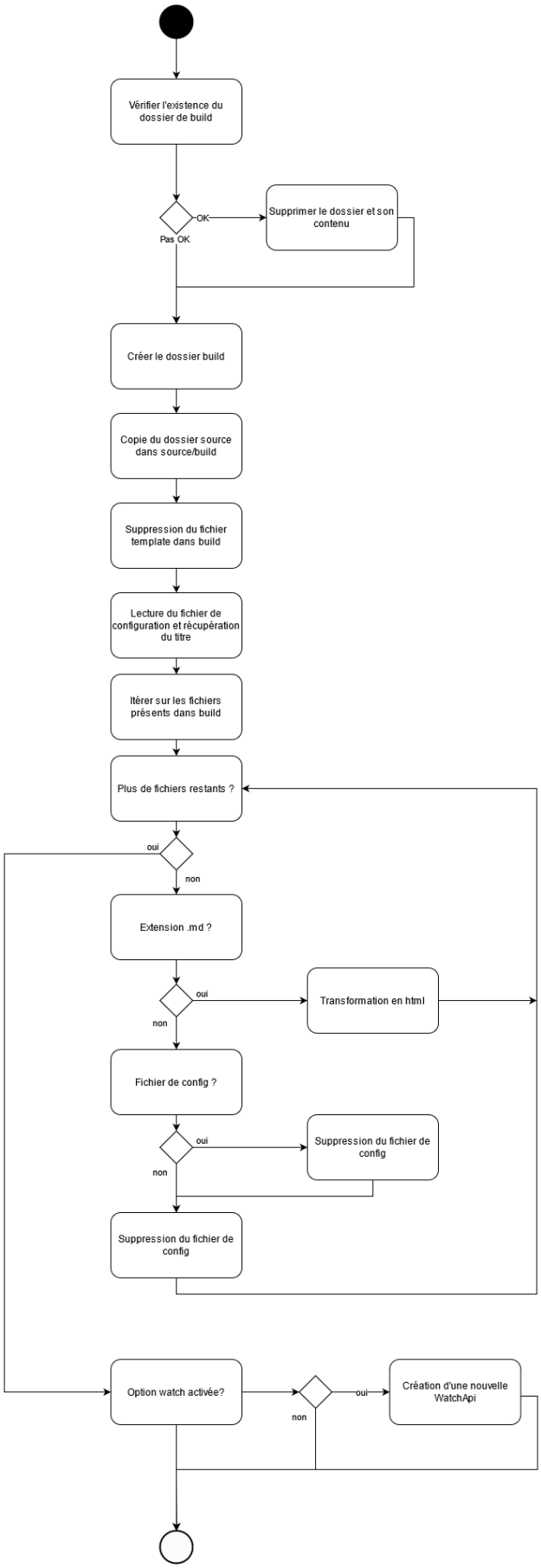


4.3 Activité

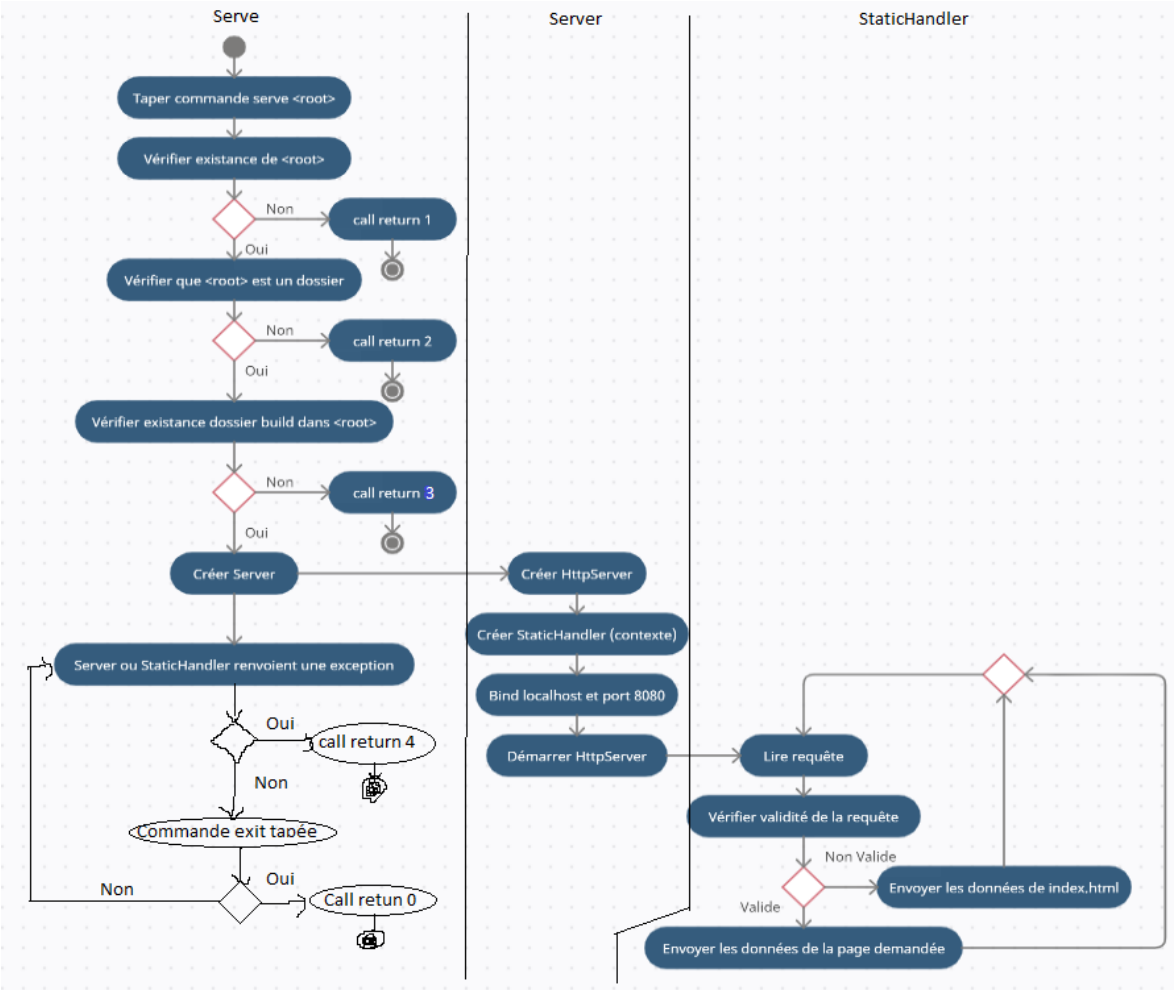
4.3.1 Commande init



4.3.2 Commande build



4.3.3 Commande Serve (sans watch API)

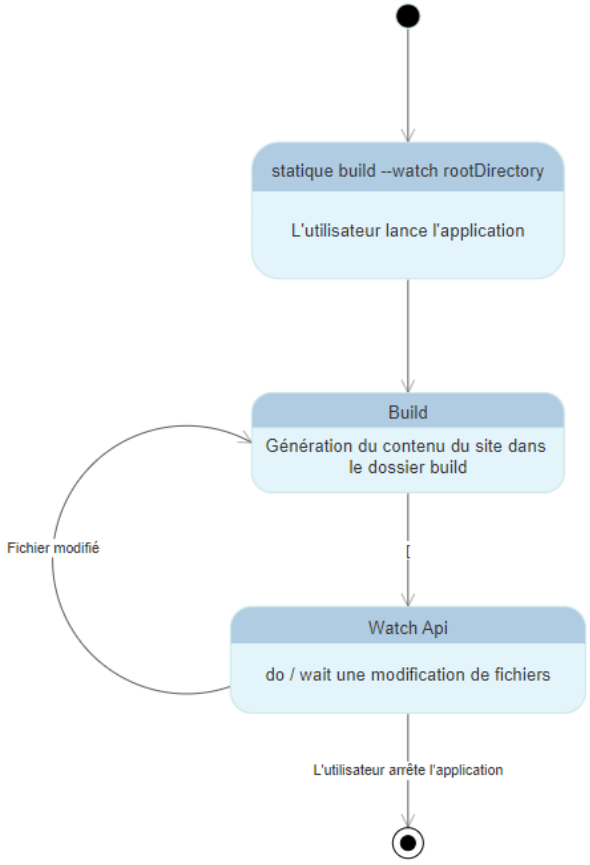


Ce diagramme a été fait avec https://app.creately.com/ et fini avec le logiciel Paint car le premier était payant à partir de 60 composants.

4.4 Etat

Diagramme de la commande Build avec watch Api

Logiciel utilisé : https://www.smartdraw.com/state-diagram/


statique build --watch rootDirectory
L'utilisateur lance l'application
Build
Génération du contenu du site dans le dossier build
Fichier modifié
Watch Api
do / wait une modification de fichiers
L'utilisateur arrête l'application

5 Implémentation

5.1 Choix technologiques

5.1.1 Fichier de configuration

Nous avons choisi du JSON car c'est un format de fichier de configuration répandu. De plus, dans le cadre du cours du SER de l'HEIG, nous avons vu comment parser du JSON.

5.1.2 Markdown à HTML

Pour transformer du markdown en HTML, nous avons choisis cette librairie : https://markdown.tautua.org

Elle présentait l'avantage d'être relativement simple à mettre en place

5.1.3 JSON-Simple

Cette librairie est simple à utiliser et nous la connaissions déjà par le cours de Sérialisation des données (SER)

5.1.4 JUnit

Pour nos tests unitaires car nous avions déjà utilisé cette librairie qui est très efficace et populaire

5.1.5 Maven

Maven nous permet de gérer les dépendances Java ainsi que de faire de nombreuses tâches utiles, telles qu'exécuter les tests unitaires, générer la Javadoc et créer les packages d'installation de notre application.

Nous aurions pu tester Gradle, mais Maven était l'alternative que nous connaissions le mieux donc il nous a paru évident d'en profiter.

5.1.6 HttpServer

Pour la commande "Serve", la librairie HttpServer a été utilisée afin de créer et exécuter le serveur.

L'utilisation de celle-ci a été choisie car elle permet de simplifier le code et qu'il existe beaucoup de tutoriels sur le net qui expliquent sa mise en œuvre et son utilisation. Nous n'avons pas trouvé d'alternative à cette librairie qui semblait nous offrir les mêmes avantages.

De plus sur internet, les utilisateurs conseillent cette dernière systématiquement.

Finalement, nous avons voulu éviter de faire un server http "from scratch" afin de gagner du temps et d'éviter d'avoir des problèmes difficiles à résoudre puisque nous n'avons jamais conçu de server http auparavant.

Source documentation : https://docs.oracle.com/javase/8/docs/jre/api/net/httpserver/spec/com/sun/net/httpserver/HttpServer.html

5.1.7 Handlebar

Pour le moteur de template, Handlebar, nous avions peu de connaissances dans ce genre d'outils et c'était celui qui était suggéré dans la consigne du sprint.

Par ailleurs, en lisant la documentation, nous avions l'impression qu'il serait simple de l'utiliser. Il y a très peu de code à écrire pour utiliser le moteur.

Malheureusement, handlebar fut le point qui nous a pris le plus de temps. La documentation s'est révélée être très floue dans de nombreux aspect, et c'est après avoir réussi à l'utiliser que nous nous rendons réellement compte à quel point elle nous paraissait cryptique. En effet, pour une vingtaine de

ligne de code, il nous a fallu (en temps cumulé) plusieurs dizaines d'heures d'essais, d'échec et de cheveux arrachés pour faire fonctionner quelque chose qui finalement n'aurait pas dû excéder quelques minutes (apprentissage exclu).

Nous n'étions pas non-plus très familiarisé avec l'utilisation des moteurs, bien que nous eussions testé quelques peu Jekyll qui utilise le moteur Liquid. Cela a probablement ralenti notre compréhension, puisque nous n'avions pas de connaissances précises sur comment fonctionnait un moteur de template.

Pour résumer, Handlebar a l'avantage de proposer des fonctionnalités demandant peu de lignes de code, et elle présente l'énorme désavantage d'avoir une documentation compliquée à utiliser.

5.2 Code

- Les variables ont leur nom sous la forme de CamelCase(ex : adresseEmail)
- Les constantes static final peuvent contenir des soulignés : ADRESSE_EMAIL
- Les fonctions doivent être commentées

5.3 Site

5.3.1 Article

Un article sous la forme d'un fichier md doit/peut contenir :

- Titre
- Auteur
- Date
- Le contenu

5.3.2 Fichier de configuration

Un fichier ce configuration contient les informations identifiants le site. L'utilisateur peut les configurer directement sur la console ou créer son propre fichier de configuration.

Les champs sont les suivants : titre, domaine, description

Il peut laisser les champs vide lors de la création.

6 Pratiques Agile

6.1 Test first programming

6.1.1 Handlebar

Pour Handlebar, nous avions implémenté un test avant d'implémenter le code

Cela nous a permis de nous poser en premier la question "Que devons-nous obtenir" avant de se demander "comment implémenter".

Ce code de test n'était pas totalement complet et était plus un template de base, qui a nous a été utile une fois le code écrit. Il était sur une branche annexe et a ensuite été mergé sur une autre branche lorsqu'on avait terminé l'implémentation de Handlebar.

Lien du commit : 8bd94f170d41326f267b6328ef1ee4607b7b23c0

Lien url du commit : https://github.com/gen-classroom/projet-pellissier_ruckstuhl_sauge_viotti-prsv/commit/8bd94f170d41326f267b6328ef1ee4607b7b23c0

6.1.2 Serve

Il n'a pas été possible de faire du "test first programming". Puisque c'est la première fois que nous implémentions un server http, il était difficile de savoir à l'avance les différents cas qui devraient être testés. Un test n'aurait eu de sens seulement quand la fonction serve a été implémenté pour vérifier s'il se lance et surtout, si ce dernier ne se lance pas et pour quelles raisons.

6.2 Amélioration de l'intégration continue

À l'aide de GitHub Actions, nous avons pu automatiser la publication des releases de l'application, à l'aide de l'Action "Automatic Releases".

Lorsqu'un nouveau tag est publié dont le nom commence par la lettre 'v' (par exemple v0.0.1), l'action va générer le package de la release avec 'mvn package' et l'inclure en tant qu'asset dans une nouvelle release.

Nous avons également pu automatiser le lancement des tests unitaires sur GitHub afin de vérifier si les PR peuvent être mergées ou non.

6.3 Automatisation de tâches

Nous avons pu automatiser beaucoup de tâches durant le projet :

- Publication automatique des releases avec génération des packages (explications au point précédent)
- Le numéro de version affiché par l'application est récupéré directement depuis les métadonnées du .jar, ce qui fait qu'il n'y a pas besoin de le mettre à jour à la main dans le code (c'est le pom.xml qui définit la version de l'ensemble du projet)
- Génération des packages d'installation avec Maven, qui contiennent entre autres :
    - .jar avec le numéro de version dans le nom du fichier
    - Librairies
    - Scripts d'exécution
    - Javadoc : générée automatiquement
    - README du projet
    - Licence du projet
- Tests unitaires automatiques par GitHub Actions lorsque la branche main est mise à jour

- Calcul automatique du code-coverage lorsque la branche main est mise à jour, le badge affiché sur le README affiche le résultat

6.4 Refactoring

Lors du sprint 3, nous avons dû refactorisé le code afin de mettre nos fichiers sources dans un package appelé `ch.heigvd.prsv`.

Cette opération était nécessaire pour faire fonctionner le benchmark.

Pour ce faire, nous avons décidé de merger d'abord nos pull request respectives puis ensuite faire le refactoring.

Ainsi, il n'y avait plus personne qui travaillait sur une branche non refactorisé, ce qui aurait eu pour conséquence de créer des conflits lors du merge avec la branche refactorisé.

Cet événement nous a fait comprendre qu'il était important de placer toutes les classes dans un package dès le début.

6.5 Commit early, commit often

Nous avons commit régulièrement nos changements sur nos branches respectives. De plus, des pull request étaient actives, par exemple sur la fonctionnalité Serve, permettant à chacun de voir l'évolution du code.

6.6 Pair Programming

Nous étions 2 personnes sur la fonctionnalité Handlebar et la WatchAPI. Nous avons alors pu faire du pair programming afin de régler les bugs et de mieux intégrer le moteur de template à la commande build.

De plus, il était plus sympa et moins décourageant de travailler à deux pour trouver des solutions.

Pour ce faire, nous avons utilisé l'outil *Partage d'écran* de Discord.

7 Tests et intégration continue

7.1 Test unitaire

Pour les tests unitaires, nous avons observés les deux principes suivants :

1) Les test étaient réalisés avant le code ou alors au tout début.
2) Si ce n'était pas possible, alors les tests étaient été faits au fur et à mesure que les fonctions étaient créées. Il y avait beaucoup de fonctions de traitements de fichiers, nous avons trouvés plus simple de d'abord écrire du code puis les tests. Néanmoins les tests étaient réalisés au fur et à mesure.

7.1.1 Description

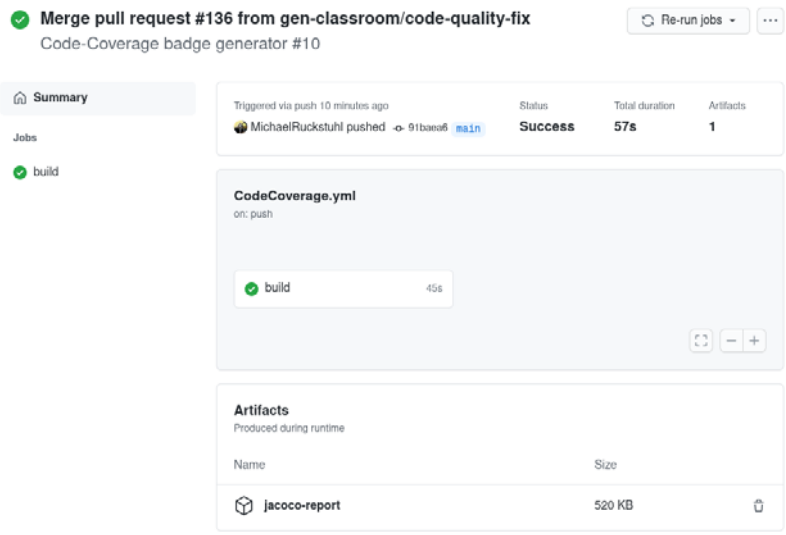
Voici un résumé des tests unitaires JUnit effectués :

- Des tests ont été effectués pour la majorité des fonctions statiques proposées par la classe FileHandler. Cette classe permet d'utiliser des fonctions sur les dossiers et fichier utilisée plusieurs fois dans les autres classes.
- L'écriture et la lecture de fichier JSON est testée. Il est également vérifié que les titres et noms de domaines sont correctes.
- Transformer fichier md -> html : il est vérifié que la classe MdToHtml puisse créer un .html dans une destination depuis une source en .md.
- Commande Init :
    - Vérifie que les fichiers de config, d'index et que dossier root existent.
    - Vérifie le contenu de index.md et du fichier de config.
- Commande build :
    - vérifie que les fichiers correspondants existent.
    - Test également de handlebar
    - Test aussi la watch api en créant un nouveau thread. Nécessite néanmoins un checkup manuel pour vérifier que les fichiers crées existent.
- Commande clean:
    - Vérifie que le programme exit 0 quand le dossier build n'est pas dans le dossier mentionné.
    - Vérifie que le programme exit 0 quand le dossier mentionné n'existe pas.
    - Lorsque le dossier mentionné existe et contient un dossier build, vérifie que le programme exit 1 et que le dossier build a bien été supprimé du dossier mentionné.
- Commande Serve : des tests ont été mis en place pour vérifier les cas limites, par exemple si aucun dossier n'est présent.

7.2 Code coverage

Nous avons configuré **Jacoco5** afin de calculer le degré de couverture de notre code par des tests unitaires et d'intégration.

Sur le README de notre repo GitHub on peut voir un badge "Coverage" qui est mis à jour après chaque commit sur la branche main. Dans GitHub Actions on peut trouver le rapport téléchargeable de Jacoco en tant qu'artefact :



À noter que le code coverage sur le main est plus faible que dans ce rapport car les tests de benchmark ont été ajoutés sur le main, mais ils peuvent être ignorés du code coverage.

Les résultats sont les suivants :

ch.heigvd.prsv

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Statique		0%		n/a	3	3	5	5	3	3	1	1
Serve.new Thread() {...}		0%		n/a	2	2	3	3	2	2	1	1
Serve		86%		77%	4	11	6	27	0	2	0	1
New		0%		n/a	2	2	3	3	2	2	1	1
Build		95%		77%	4	13	6	65	0	4	0	1
Constantes		0%		n/a	1	1	1	1	1	1	1	1
Init		100%		100%	0	13	0	93	0	6	0	1
Clean		100%		100%	0	6	0	9	0	2	0	1
Total	103 of 831	87%	8 of 58	86%	16	51	23	205	8	22	4	8

Nous pouvons constater que les commandes de l'application sont globalement très couvertes par les tests. Nous voyons que très peu de méthodes instructions, branches ou lignes de codes ne sont pas couvertes.

Serve

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
call()		84%		77%	4	10	6	24	0	1
Serve()		100%		n/a	0	1	0	3	0	1
Total	15 of 108	86%	4 of 18	77%	4	11	6	27	0	2

Build

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
call()		94%		77%	4	10	6	64	0	1
lambda$call$0(Path)		100%		n/a	0	1	0	1	0	1
Build()		100%		n/a	0	1	0	1	0	1
lambda$call$1(Path)		100%		n/a	0	1	0	1	0	1
Total	13 of 269	95%	4 of 18	77%	4	13	6	65	0	4

Il est compliqué de faire davantage de tests sur la fonction call des deux classes ci-dessus en pratique. Elle en déjà été bien testées. En faire plus nous ferais perdre du temps pour très peu de gains et complexifierait le code.

ch.heigvd.prsv.utils

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
WatchApi.WatchApiRegister		72%		37%	8	13	9	38	1	5	0	1
FileHandler		75%		91%	2	14	14	48	1	8	0	1
Version		0%		n/a	2	2	5	5	2	2	1	1
JSONConfig		89%		n/a	0	9	6	40	0	9	0	1
MdToHtml		73%		50%	2	3	7	19	1	2	0	1
HandlebarUtil		90%		n/a	0	3	3	15	0	3	0	1
Contenu		94%		62%	3	6	1	20	0	2	0	1
WatchApi		87%		n/a	0	2	3	11	0	2	0	1
WatchApi.WatchApiRegister.new SimpleFileVisitor() {...}		100%		100%	0	3	0	7	0	2	0	1
Total	162 of 749	78%	15 of 40	62%	17	55	48	202	5	35	1	9

Dans le package utils, à l'exception des classes pour watchApi , le même constat est possible. Pour la version, les fonctions sont simples et pourtant difficiles à tester.

Pour Contenu, Jacobo est particulièrement sévère car nous ne testons pas deux branches peu sujettes à des fautes.

En ce qui concerne FileHandler, les fonctions ont toutes été testées à au moins 50%. Nous avons également fait de notre mieux pour tester le nécessaire sans que cela ne devienne pas inutilement compliqué.

Pour WatchApi, il convient de préciser que le problème selon le logiciel vient surtout des branchs. Cependant, beaucoup de conditions serait inutilement compliquées à tester et nous avons besoin d'une boucle while infinie pour faire fonctionner le code, ce qui fausse le résultat.

ch.heigvd.prsv.utils.server

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
StaticHandler		26%		22%	22	30	79	101	2	6	0	1
Server		87%		n/a	1	2	1	6	1	2	0	1
StaticHandler.Asset		100%		n/a	0	1	0	3	0	1	0	1
Total	322 of 464	30%	37 of 48	22%	23	33	80	110	3	9	0	3

Ici également, le résultat donne l'impression que le server n'est pas testé. Cependant, les points critiques sont globalement testés. Par rapport au temps que nous avions à disposition, tester l'entièreté du serveur aurait été difficile et inutilement compliqué. De plus le code est peu modulable et donc prédictible.

7.2.1 Conclusion

Jacoco est un outil très utile pour détecter où nous devrions faire des efforts supplémentaires. Cependant, il ne faut pas forcément s'affoler si les barres rouges sont grandes et garder un esprit critique sur ce qu'il est utile de tester par rapport au budget temps.

Sources utilisées :

Lien :

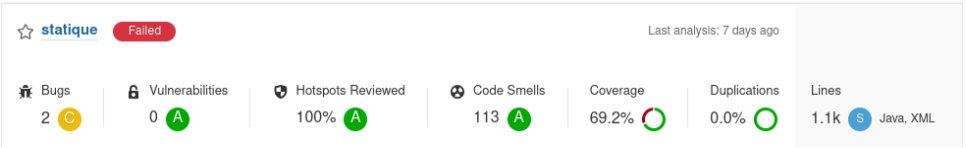
- https://medium.com/capital-one-tech/improve-java-code-with-unit-tests-and-jacoco-b342643736ed
- Exemples complets : https://mkyong.com/maven/maven-jacoco-code-coverage-example/
- GitHub Actions: https://github.com/marketplace?query=jacoco
- Maven repo : https://mvnrepository.com/artifact/org.jacoco/jacoco-maven-plugin

7.3 Code quality

Nous avons utilisé **SonarQube7** pour détecter des bugs et des vulnérabilités dans notre code.

Nous avons pu détecter des erreurs dans notre code, comme le fait que nos fichiers n'étaient pas correctement fermés en cas d'erreur.

Voici nos résultats finaux sur la qualité du code :



SonarQube marque ce projet en tant que “failed” à cause du Code coverage qui est en dessous de 80%. Cependant, le code coverage prend en compte tout le code du projet, dont la classe Statique et le code du serveur HTTP qui sont difficilement testables avec des tests unitaires.

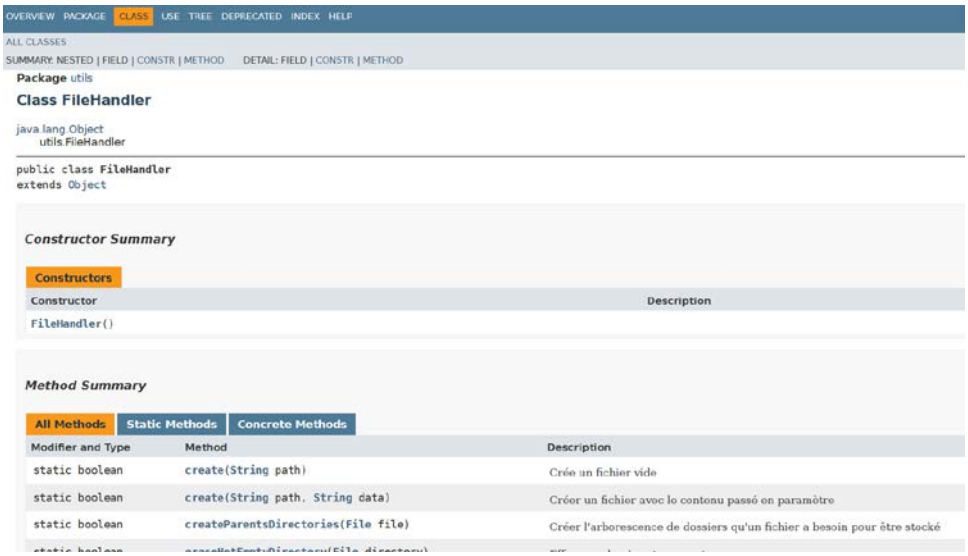
Il y a beaucoup de “Code smells”, mais étant donné le nombre de fautes et leur importance moindre par rapport aux autres tâches nous n’avons pas corrigé ce point.

Nous prendrons cependant en compte les Code smells pour nos futurs codes java. (à part le fait de déclarer les variables avec “var”, on est plutôt d’avis qu’il faut donner explicitement le type).

7.4 Documentation

La génération de Javadoc est possible grâce au plugin maven-javadoc-plugin. La génération se fait grâce à la commande mvn javadoc:javadoc et aussi lors d’un mvn package ou un mvn install. Le dossier API doc est ajouté dans la release grâce à assembly.xml.

Dans le dossier apidoc de la release, il est possible d’ouvrir la javadoc en format HTML:



8 Benchmark

8.1 JMH

Nous avons lancé un benchmark avec JMH sur la commande build. Le contenu à build est celui par défaut créer par la commande init.

8.1.1 Mode Throughput

Le mode utilisé est thrpt, qui mesure le nombre d'opération par secondes. Plus celui-ci est élevé, mieux c'est. C'est difficile de réaliser une interprétation sur ces résultats car nous n'avons pas de points de comparaison

```
Result "ch.heigvd.prsv.benchmark.BenchmarkRunner.init":
  16,458 ±(99.9%) 1,837 ops/s [Average]
  (min, avg, max) = (11,284, 16,458, 20,083), stdev = 2,452
  CI (99.9%): [14,621, 18,295] (assumes normal distribution)


# Run complete. Total time: 00:08:27

REMEMBER: The numbers below are just data. To gain reusable insights, you need to follow up on
why the numbers are the way they are. Use profilers (see -prof, -lprof), design factorial
experiments, perform baseline and negative tests that provide experimental control, make sure
the benchmarking environment is safe on JVM/OS/HW level, ask for reviews from the domain experts.
Do not assume the numbers tell you what you want them to tell.

Benchmark              Mode  Cnt   Score   Error  Units
BenchmarkRunner.init  thrpt   25  16,458 ± 1,837  ops/s
```

8.1.2 Mode Average Time

Mesure le temps moyen d'exécution de la commande build

```
Result "ch.heigvd.prsv.benchmark.BenchmarkRunner.init":
  0,059 ±(99.9%) 0,003 s/op [Average]
  (min, avg, max) = (0,051, 0,059, 0,068), stdev = 0,004
  CI (99.9%): [0,056, 0,062] (assumes normal distribution)


# Run complete. Total time: 00:08:28

REMEMBER: The numbers below are just data. To gain reusable insights, you need to follow up on
why the numbers are the way they are. Use profilers (see -prof, -lprof), design factorial
experiments, perform baseline and negative tests that provide experimental control, make sure
the benchmarking environment is safe on JVM/OS/HW level, ask for reviews from the domain experts.
Do not assume the numbers tell you what you want them to tell.

Benchmark             Mode  Cnt  Score   Error  Units
BenchmarkRunner.init  avgt   25  0,059 ± 0,003   s/op
```

Sources :

- https://soat.developpez.com/tutoriels/java/comprendre-fonctionnement-jmh/
- http://tutorials.jenkov.com/java-performance/jmh.html#jmh-benchmark-modes

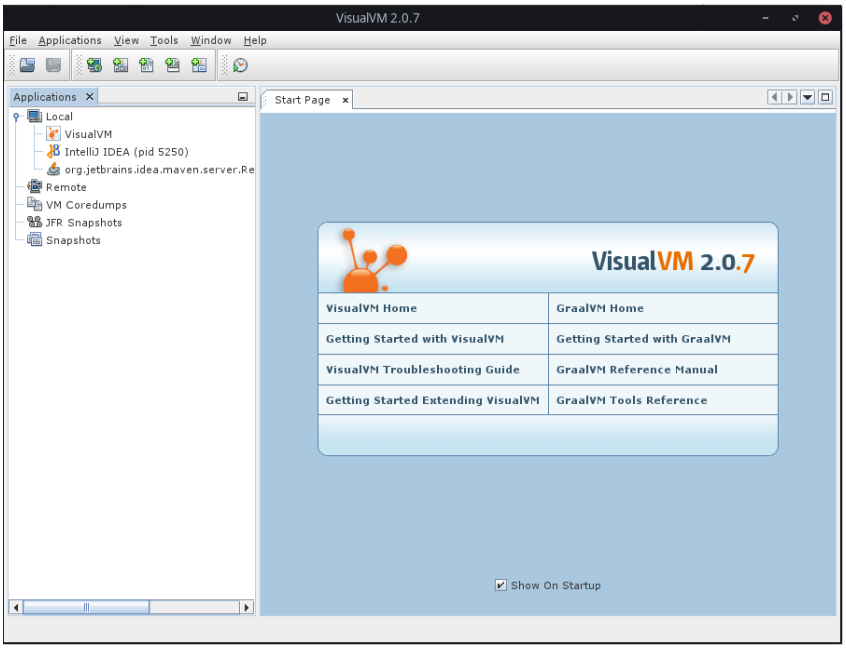
8.2 VisualVM9

8.2.1 Tutoriel d'utilisation

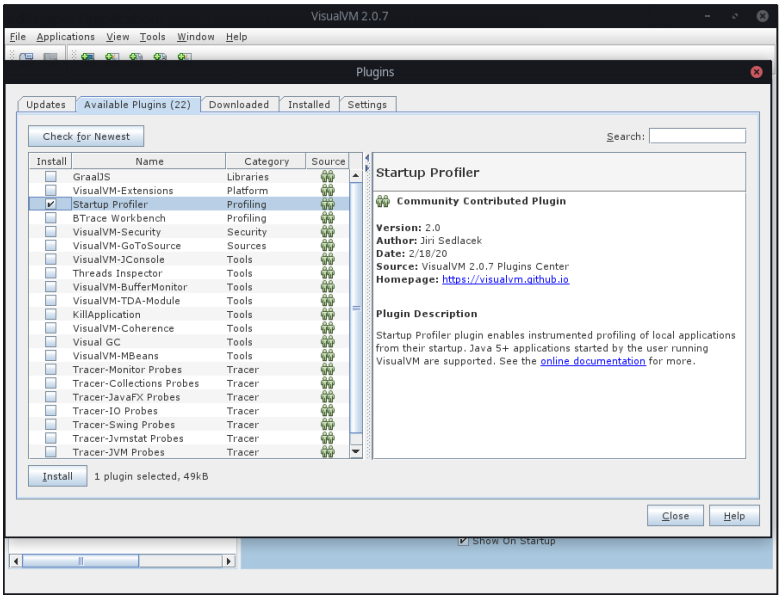
Liens utiles:

- https://visualvm.github.io/startupprofiler.html

1. Télécharger et dézipper l'application: https://github.com/oracle/visualvm/releases/download/2.0.7/visualvm_207.zip
2. Exécuter l'application (.exe ou .sh) qui se trouve dans le répertoire /bin de VisualVM

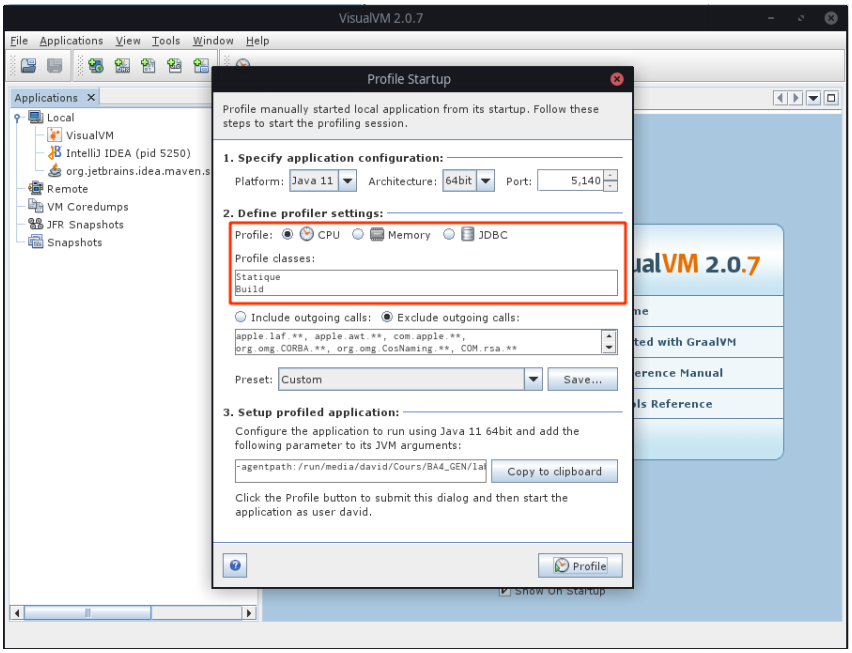


3. Dans Tools->Plugins, installer le plugin "Startup Profiler"

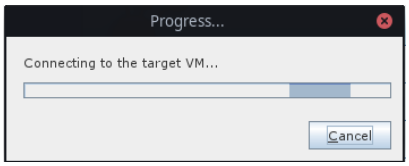


4. Dans le menu Applications : Ouvrir "Profile Startup"

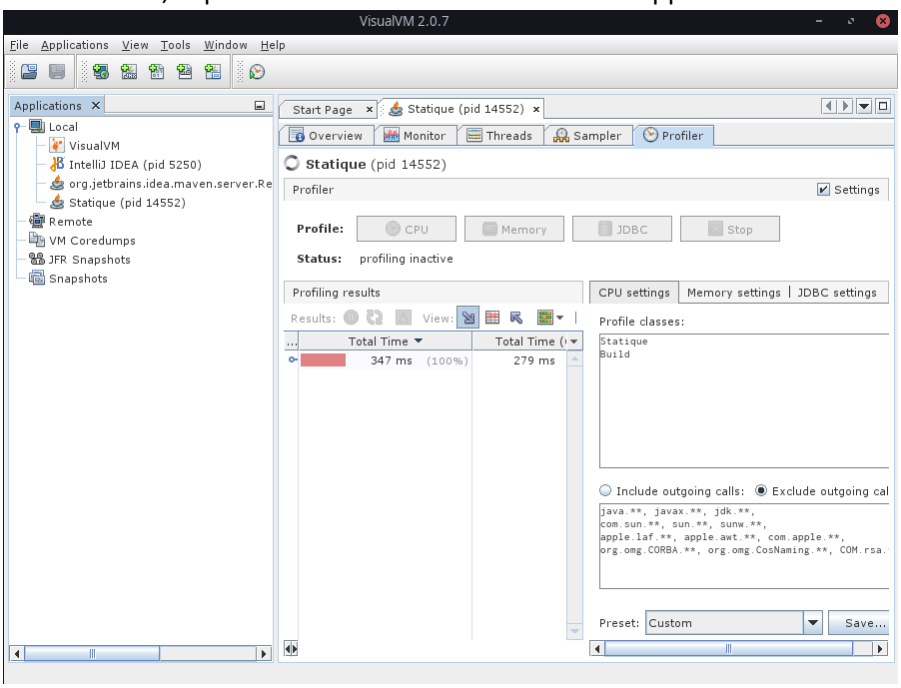
5. Définir les options du profiler



6. Dans la partie 3 de la fenêtre, copier l'argument donné et cliquer sur Profile. VisualVM se met en attente

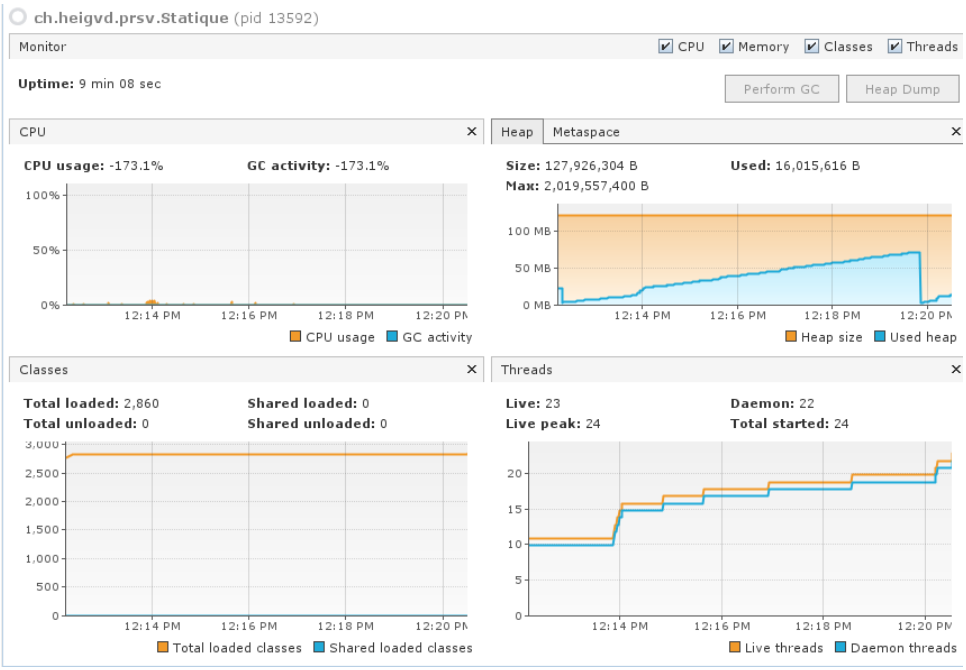


7. Exécuter l'application tout en donnant l'argument qu'on a copié juste avant
    a. Exemple sous Linux: java <-agentpath:…> -cp "../lib/*" Statique
8. Sur VisualVM, le profiler se lance dès l'exécution de l'application



8.2.2 Résultats du benchmark

Sur une exécution de la commande "statique build --watch"



On peut voir que notre application consomme très peu en CPU (il y a de minuscules pics lorsqu'il y a un nouveau build), et que la taille de la mémoire heap utilisée est en dessous de 50 Mo.

Nous ne savons cependant pas comment interpréter les autres valeurs.

9 Conclusion

C'est ainsi que nos fiers aventuriers de l'équipe PRSV menèrent à bien la noble tâche confiées. Ils ressortirent fatigués de cette aventure mais heureux d'avoir accompli cette difficile tâche.

Le soir au coin du feu, les yeux brillants de cette incroyable aventure, ils reparlent encore de cette riche expérience leur ayant appris à mettre en place un processus Agile et des leçons qu'ils ont pu en tirer

1) Avec les pull requests, ils apprirent que tout travail mérite d'être vérifié par une seconde personne afin de pouvoir donner un avis constructif.
2) Avec les commits, ils apprirent à décrire leur action pour que quiconque comprenne leur agissements.
3) Comme dirait le célèbre sage Shun Tzu "*Tout le succès d'une opération réside dans sa préparation*". L'utilisation d'un Kanban et des issues a permis de planifier minutieusement chaque sprint afin d'atteindre le succès.
4) En transformant du markdown à HTML, ils firent leur début en tant qu'alchimiste et comprirent que de trouver une bonne librairie vaut mieux que de de réinventer la roue.
5) Face à l'adversité, ils firent l'expérience du travail en duo pour triompher des nombreux adversaires rencontrés sur leur chemin tel que la Watch Api et Handlebar.
6) Pour améliorer leur sécurité et leur équipement[ndlr : *code]*, le grand sage SonarQube leur prodigua maints conseils.
7) Afin que leur aventure ne tombe pas dans l'oubli après leur mort, ils apprirent à générer automatiquement de la documentation. Leur histoire restera à jamais gravé dans le HTML5.

Pour conclure, le plus important n'est pas le trésor qui attend au bout du chemin, mais l'expérience acquise tout au long du voyage.