**Rapport - Groupe B1-2**

Projet SOA du projet intégrateur

*Lien vers le code source :* [*https://github.com/vsaulnie/SOA\_PI\_Community\_Detection.git*](https://github.com/vsaulnie/SOA_PI_Community_Detection.git)

**DECAESTECKER - DHOUIB - DIARRA - SAULNIER - NEDU**

**Table des Matières**

[**Introduction**](#_heading=h.p9s38fjhasji) **2**

[**Méthodes, Outils et Technologies**](#_heading=h.15fnbn3jwq7t) **2**

[I. Partie IL](#_heading=h.c0a54q8srsp6) 2

[Méthode SCRUM & Jira](#_heading=h.ml3w0gazoqhu) 2

[GitHub](#_heading=h.g89ekun888ic) 2

[CI/CD & Jenkins](#_heading=h.c7wyehulglbx) 3

[II. Partie SOA](#_heading=h.4qq20jnq07rb) 3

[Serveurs Spring Boot](#_heading=h.mwfcj8adrsrx) 3

[API REST](#_heading=h.k7tx9dfdpc7) 3

[Serveur Découverte Eureka](#_heading=h.2saszd7mz409) 3

[**Réalisation et organisation**](#_heading=h.act00fquxn8r) **4**

[I. Identification du périmètre](#_heading=h.enegvutmha6y) 4

[I.1. Identification d’un scénario de bout en bout](#_heading=h.cshdhyye9j18) 4

[I.2. Identifier les besoins métiers (business capabilities)](#_heading=h.uovon2t8eoev) 4

[I.3. Identifier les micro-services découlant des besoins métiers](#_heading=h.2o7yq1uh9jk7) 4

[I.4. Identifier l’architecture globale](#_heading=h.jj2mjd6e7tgf) 5

[II. Spécification des user-stories](#_heading=h.qodfwou75px4) 5

[III. Planification des sprints sur JIRA](#_heading=h.bppekamsu440) 6

[IV. Réalisation des sprints](#_heading=h.4mdysyq9n1cf) 7

[Sprint 1 : Mise en place de l’environnement et de l’intégration continue](#_heading=h.ifgn4ufya5jq) 7

[Sprint 2 : Formatage de Graphe et Microservice Giraph](#_heading=h.tuvgne5mzqja) 8

[Sprint 3 : Interaction avec les solutions](#_heading=h.tfadbado9agi) 9

[Diagramme de Gantt du projet](#_heading=h.itvxeuh8hqib) 10

[**Conclusion**](#_heading=h.h8g3ovfhk9ub) **11**

[Objectifs réalisés](#_heading=h.fwnjxlwypzkq) 11

[Objectifs non-réalisés](#_heading=h.81ae4dis9h71) 11

[Perspectives d’amélioration](#_heading=h.v82tuvoke0u8) 11

[Acquis](#_heading=h.66wqiajnri6t) 11

# Introduction

Durant notre projet intégrateur, nous avons mené une étude comparative de deux solutions de stockage de graphes (Neo4J et Apache Giraph) dans le cadre de la détection de communauté dans les grands graphes. Ayant dégagé les avantages et limites respectives de ces deux solutions, nous avons, durant une seconde phase de ce projet, développé une architecture orientée microservice permettant d’offrir un accès commun à ces plateformes et de les exploiter simultanément, afin de poursuivre la comparaison ou bien, exploiter la plus adaptée selon le besoin.

# Méthodes, Outils et Technologies

## I. Partie IL

### Méthode SCRUM & Jira

Durant ce projet, nous avons travaillé selon une méthode Agile : la méthode SCRUM, travaillant notamment par sprints itératifs après définition de “User Stories” plus ou moins prioritaires.

Pour définir la charge de travail de chaque user story définie dans la backlog JIRA. Nous avons utilisé la méthode Poker, par vote grâce scrumpoker.online. Ensuite, nous avons fixé la note finale sur la base d’un consensus.

Le détail du déroulé de cette méthode SCRUM et de nos sprints est présenté dans la deuxième partie de ce rapport.

Afin de gérer l’application de cette méthode, nous nous sommes reposés sur l’outil JIRA, qui offre de nombreuses fonctionnalités liées aux méthodes Agile : backlog, définition et suivi des sprints, etc...

### GitHub

Chacun des microservices identifiés s’est vu attribuer sa propre branche sur un dépôt Git partagé (voir en-haut de ce document). Ainsi, les membres développant un microservice ont pu mener leur développement de manière isolée, tout en bénéficiant des mises à jour concernant les ressources communes.

La branche *main* contient des fichiers communs au projet (documentation, scripts…). Les autres branches contiennent les fichiers propres à son micro-service.

### CI/CD & Jenkins

Dans une volonté d’automatiser notre travail du développement au déploiement suivant les principes de Continuous Integration-Continuous Development, nous avons créé un projet Jenkins par microservice. Chaque projet Jenkins est lié avec sa branche GitHub respective. A la suite d’un commit sur une branche liée à un microservice du dépôt git, Jenkins va déclencher un build et générer un nouveau fichier WAR pour le projet. Afin de ne pas surcharger nos machines(qui auraient dû supporter plusieurs serveur spring + les conteneurs giraph et neo4j) ni surexploiter le Cloud OpenStack mis à notre disposition, nous n’avons pas automatisé le déploiement des microservices par Jenkins, conservant un déploiement manuel pour les tests.

Nous avons toujours gardé en tête l’idée d’avoir une pipeline Jenkins individualisée pour chaque microservice.

## II. Partie SOA

### Serveurs Spring Boot

Nous avons utilisé le framework Spring pour la majorité de nos microservices. Il présente un écosystème riche et de nombreux utilitaires dédiés à la conception de microservice. Un serveur Spring offre de plus simplicité et polyvalence de déploiement : entièrement contenu dans une archive jar auto-suffisante, on peut l’exécuter sur toute machine dotée d’une JVM. L’application déploie un serveur Tomcat par défaut au lancement, simplifiant là encore son déploiement.

### API REST

Pour les APIs de nos microservices, nous avons choisis de développer des APIs REST. En plus d’être parfaitement intégré par le framework Spring, ce formalisme offre une grande ergonomie d’utilisation et de spécification, notamment grâce à un écosystème riche dont font partie les outils Postman (pour le test) ou l’OpenApi Initiative (anciennement Swagger, pour la spécification de l’API).

### Serveur Découverte Eureka

Pour notre micro service de découverte, nous avons utilisé la technologie de serveur Eureka. Initialement développé par Netflix puis intégré au framework Spring, ce serveur offre toutes les fonctionnalités de découverte attendues, permettant à nos microservices de déléguer les problématiques de routage ou de configuration des points d’accès, et concentrant leur développement sur la logique applicative. Dans un cadre de l’intégration et développement continus, cela permet de faciliter le déploiement des services. Pour des perspectives futures, cela ouvre la voie à des possibilités d’équilibrage de charge ou de maintenance.

# Réalisation et organisation

Présentons maintenant le déroulement du projet, de manière chronologique depuis notre définition des besoins et architectures jusqu’à la réalisation des sprints.

## I. Identification du périmètre

### I.1. Identification d’un scénario de bout en bout

L’utilisateur de l’application micro-service se connecte via son navigateur web sur un front-end springboot (qui a accès à tous les micro services).

Il choisi dans une liste déroulante le graphe qu’il veut exploiter. Il peut également donner l’url d’un graph (au format txt ou compressé gz) qu’il souhaite ajouter. (Les graphes sont traduits pour chaque système exploitée et disponibles dans un micro-service dédié).

Il sélectionne dans une liste déroulante l’algorithme qui va être lancé sur le graphe (l’algorithme doit être implémenté dans tous les systèmes (Neo4J, Giraph..) .

Il peut visualiser les systèmes disponible et sur lesquelles vont être lancé le couple Graphe-Algo.

Il peut lancer avec un bouton la comparaison entre les solutions.

Lorsqu’une solution donne sa réponse, elle est affichée dans un cadre dédié (tableau, console?) .

Un dernier bouton peut permettre d’exporter au format XML ? HTML le résultat.

Il peut également consulter des requêtes déjà réalisées dans la passé.

### I.2. Identifier les besoins métiers (business capabilities)

1. Lancer des requêtes sur les diverses solutions
2. Importer des graphes dans les diverses solutions (téléchargement & formatage)
3. Collecter les résultats de requêtes
4. Consulter des résultats de requêtes
5. DIaloguer avec Neo4J : OK (API HTTP intégrée)
6. Dialoguer avec Giraph : Implémenter notre propre API
7. Pouvoir ajouter autant de solutions que désiré
8. Initier les ordres d’exécution depuis une interface graphique
9. Afficher les résultats des requêtes depuis une interface graphique

### I.3. Identifier les micro-services découlant des besoins métiers

- **[Launcher]** Service de gestion des systèmes : chargement des graphes + traduction de format + envoi de requêtes → *business capabilities 1 & 2*

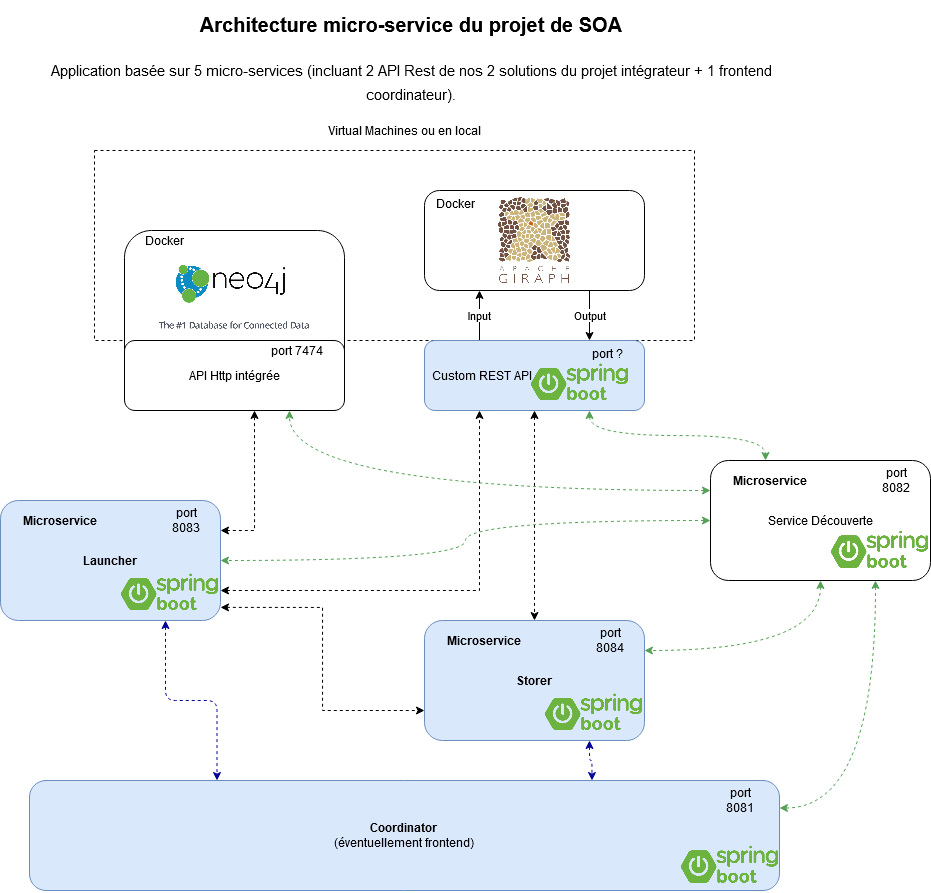
- **[Storer]** Service de stockage des résultats → *business capabilities 3 & 4*

- **[Giraph][Neo4J]** API Giraph custom & API Neo4J → *business capabilities 5 & 6*

- **[Discovery]**Service découverte → *business capabilities 7*

- **[Frontend]** Service frontend avec capacité de coordination → *business capabilities 8 & 9*

### I.4. Identifier l’architecture globale



## II. **Spécification des user-stories**

Définir un ensemble de fonctionnalités (classées dans l’ordre de priorité)

1. **Prendre en entrée un graphe au format txt ou gz et être capable de le traduire au bon format pour Neo4J et Giraph**

Estimation d’effort (méthode poker) : 10

1. **Ajouter des graphes traduits aux bon endroits sur les Docker de Neo4J et Giraph**

Estimation d’effort (méthode poker) : 4

1. **Lancer des exécutions d’algorithmes choisis sur Neo4J et Giraph**

Estimation d’effort (méthode poker) : 4

1. **Récupérer les résultats des exécutions dans une base de donnée (ou pseudo base)**

Estimation d’effort (méthode poker) : 6

1. **Charger des graphes via url local ou web**

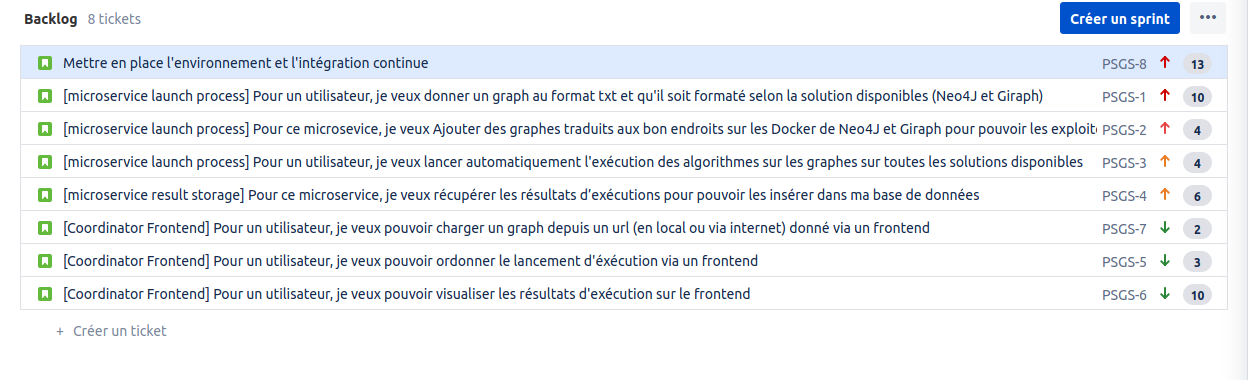
Estimation d’effort (méthode poker) : 2

1. **Initier les ordres de lancement d’exécution directement depuis le frontend**

Estimation d’effort (méthode poker) : 3

1. **Afficher les résultats d’exécution sur le frontend**

Estimation d’effort (méthode poker) : 10

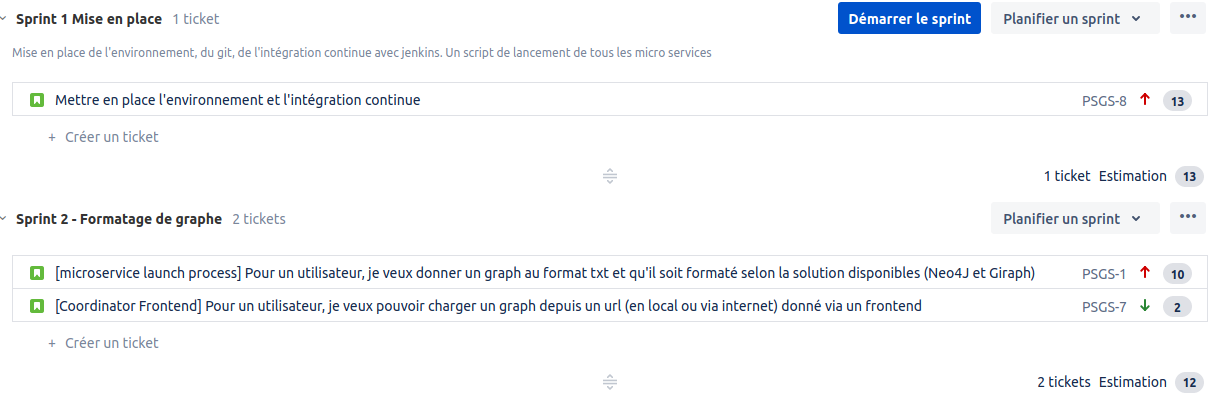
Illustration 1 : Exemple de notre 1ère backlog sur Jira

## III. **Planification des sprints sur JIRA**

Lien vers notre Jira : <https://decaeastecker-etud.atlassian.net/secure/RapidBoard.jspa?projectKey=PSGS>

2 sprints initiaux :

* 1 sprint pour la mise en place de l’environnement, la mise en place de l’intégration continue..
* 1 sprint pour les user stories les plus prioritaires ( pour le formatage des graphes et l’API Giraph)

1 sprint supplémentaire pour les interactions avec les solutions du projet intégrateur ainsi que la collection des résultats d’exécution.

## IV. Réalisation des sprints

### Sprint 1 : Mise en place de l’environnement et de l’intégration continue

Objectif : Mise en place du projet, du git, de jenkins...

Répartition des tâches :

* Brice : création des micro-services spring-boot
* Emna – Richard : Organisation git
* Assa – Valentin : Configuration Jenkins

Bilan/ conclusion :

Ce sprint est dédié à la mise en place du projet de SOA. Après avoir effectué l’ensemble du travail préliminaire, allant de l’identification du périmètre au peuplement de la backlog sur JIRA, nous avons commencé la réalisation du 1er sprint.

Les micro-services identifiés dans l’architecture de base ont tout d’abord été générés avec Spring Initializr. Chaque micro-service a été nommé pour pouvoir être découvert par notre service de découverte Eureka et s’est vu attribuer un port de déploiement. Le micro-service de client configuration est le seul à ne pas fonctionner. En effet, il ne parvient pas à récupérer les fichiers présents sur le git relié au serveur de configuration. Nous en avons notifié Mme. Ghada et dans l’attente, nous utiliserons le fichier application.properties.

Nous avons créé un git pour le projet de SOA : <https://github.com/vsaulnie/SOA_PI_Community_Detection.git>

Le répertoire est divisé en branches, avec une branche par micro-service. La branche main contient les fichiers du projet qui ne concernent pas spécifiquement un seul micro-service. La division en branche facilite l’intégration continue avec Jenkins.

Sur Jenkins, chaque micro-service a son projet. Chaque projet est relié à une branche du git principal. Sur changement d’une branche du git, le micro-service est recompilé et un WAR est recréé.

Pour le lancement, nous préférons ne pas déployer automatiquement pour l’instant pour ne pas avoir des serveurs qui tournent en permanence pendant le développement. En effet, nous avons constaté un ralentissement important de nos PCs ou VMs pour un déploiement local, et cela représenterait un coup inutile et prohibitif sur les serveurs openstack à notre disposition. Nous optons donc pour la solution de lancer manuellement les micro-services.

Les projets étant indépendants, chacun possède son propre pipeline Jenkins sans aucune relation aux autres projets.

A voir plus tard si l’on a besoin d’ajouter des tests unitaires à la batterie JUnit présente par défaut dans les projets Spring Boot.

### Sprint 2 : Formatage de Graphe et Microservice Giraph

Objectif : Je veux pouvoir donner à Neo4J ou Giraph un graph d’entrée au bon format.

Répartition des tâches :

* Brice : réalisation de l’API Giraph + traduction de graph en format jlld (format custom).
* Emna – Richard : méthodes de communication avec l’API Neo4J
* Assa – Valentin : réalisation du chargeur de graph depuis url

Bilan/ conclusion :

API REST Giraph : fonctionne tel un micro-service. Ce microservice est déployé sous forme d’un docker (dossier infra/giraph) contenant une installation hadoop, une librairie Giraph(jar) personnalisée avec les algorithmes du projet intégrateur, et un serveur Spring offrant une interface REST pour masquer les appels système à hadoop.

Communication Neo4j : Nous allons utiliser l’API HTTP de Neo4J (déployée automatiquement sur le port 7474). En effet l’API Rest originale de Neo4J est dépréciée. Nous utiliserons la requête POST “/commit”, permettant d’écrire une ou plusieurs requêtes Cipher (similaire au SQL pour Neo4J) dans le body. Pour charger les graphes, nous utiliserons le module LOAD CSV de Neo4j en lui passant l’URL des graphes stockés en local (c’est pour offrir un point d’accès à ce module que nous avons créé “/exposeCSV”)

Launcher : Il est possible de charger un graphe depuis n’importe quelle URL Web. Le fichier contenant le graphe doit être au format texte ou compressé GZ. Le graphe est téléchargé par le launcher, il le formate et stocke ce fichier en local dans le dossier local jlld/ ou csv/.

### Sprint 3 : Interaction avec les solutions

Objectif : Je veux pouvoir envoyer des requêtes et récupérer les résultats des requêtes

Répartition des tâches :

* Brice : Permettre l’upload de graph dans Giraph + gérer les résultats d'exécution Giraph/Storer
* Emna – Richard : Permettre d’envoyer des requêtes à Neo4J et Giraph + gestion des résultats d’exécution de Neo4j
* Assa – Valentin : Consultation des entrées dans storer + Upload de graph Neo4j

Bilan/ conclusion :

La chaîne d’interactions élémentaires présentée ci-dessous depuis le téléchargement d’un graphe jusqu’à la récupération des résultats fonctionne.



### Diagramme de Gantt du projet



# Conclusion

## Objectifs réalisés

Notre architecture propose un ensemble de services cohérents capables de fonctionner en interaction pour mener de bout en bout un test sur les plateformes Neo4J et Giraph, depuis le chargement du graphe jusqu’à l’exécution d’un algorithme et la récupération des résultats.

## Objectifs non-réalisés

Aucun frontend n’a été développé, ni service permettant un accès unifié aux services launcher et storer. Nous avons utilisé postman pour effectuer nos tests.

## Perspectives d’amélioration

Outre le développement d’un frontend susmentionné, notre architecture pourrait être divisée plus finement, notamment au niveau du service launcher. On pourrait ainsi séparer les fonctionnalités liées au chargement de graphe(en les déléguant par exemple à un nouveau service loader) de celles liées aux interactions avec les plateformes. De plus, le déploiement des services Giraph et Neo4J pourrait être amélioré, en leur offrant par exemple de véritables clusters de machines à exploiter puisqu’il s’agit de technologies distribuables.

## Acquis

Finalement, dans le cadre de ce projet intégrateur, nous avons pu intégrer des compétences de conception d’architecture microservice, de gestion de projet via des méthodes agiles, et de CI/CD. Nous avons également pu retravailler et conforter nos compétences de conception et de formalisation logicielle inculquées lors d'années précédentes, et appliquer les technologies de déploiement virtualisées.