Rapport de Projet SmartLogger

Ce document a pour but de retracer l'avancement réalisé, depuis l'étape de spécification jusqu'à la dernière soutenance, signant la fin du projet.

Il vise à fournir un point de vue global sur le projet, ses enjeux, sa technicité, et sur les nombreux apports qu'il a pu nous apporter au fil des mois.

SOMMAIRE

I) Le projet SmartLogger

1) Origine et besoin client

Le projet SmartLogger a été proposé par l'entreprise Saagie, une start-up spécialisée dans le big-data, qui dans le cadre de ses activités, doit effectuer de nombreuses opérations de monitoring et de maintenance, sur les différents services web déployés par l'entreprise. A cet effet, la société dispose de logs en provenance de ces applicatifs, qui permettent de représenter leur état à un instant donné, au moyen d'une multitude d'informations différentes.

Néanmoins, le volume de données correspondant est trop important pour pouvoir être traités correctement par des opérateurs. Il leur faut donc, un système capable de surveiller l'état de ces applicatifs et d'alerter les personnes impliquées en cas de défaillance : C'est le but du projet SmartLogger.

Le projet consiste à réaliser une telle application, dont les capacités d'analyse seraient basées sur des algorithmes de Machine Learning. L'intérêt étant que le système puisse apprendre et peaufiner son modèle de prédiction, au fur et à mesure des analyses effectuées, dans le but d'augmenter constamment le taux de pertinence de ses levées d'alerte.

2) Spécifications fonctionnelles

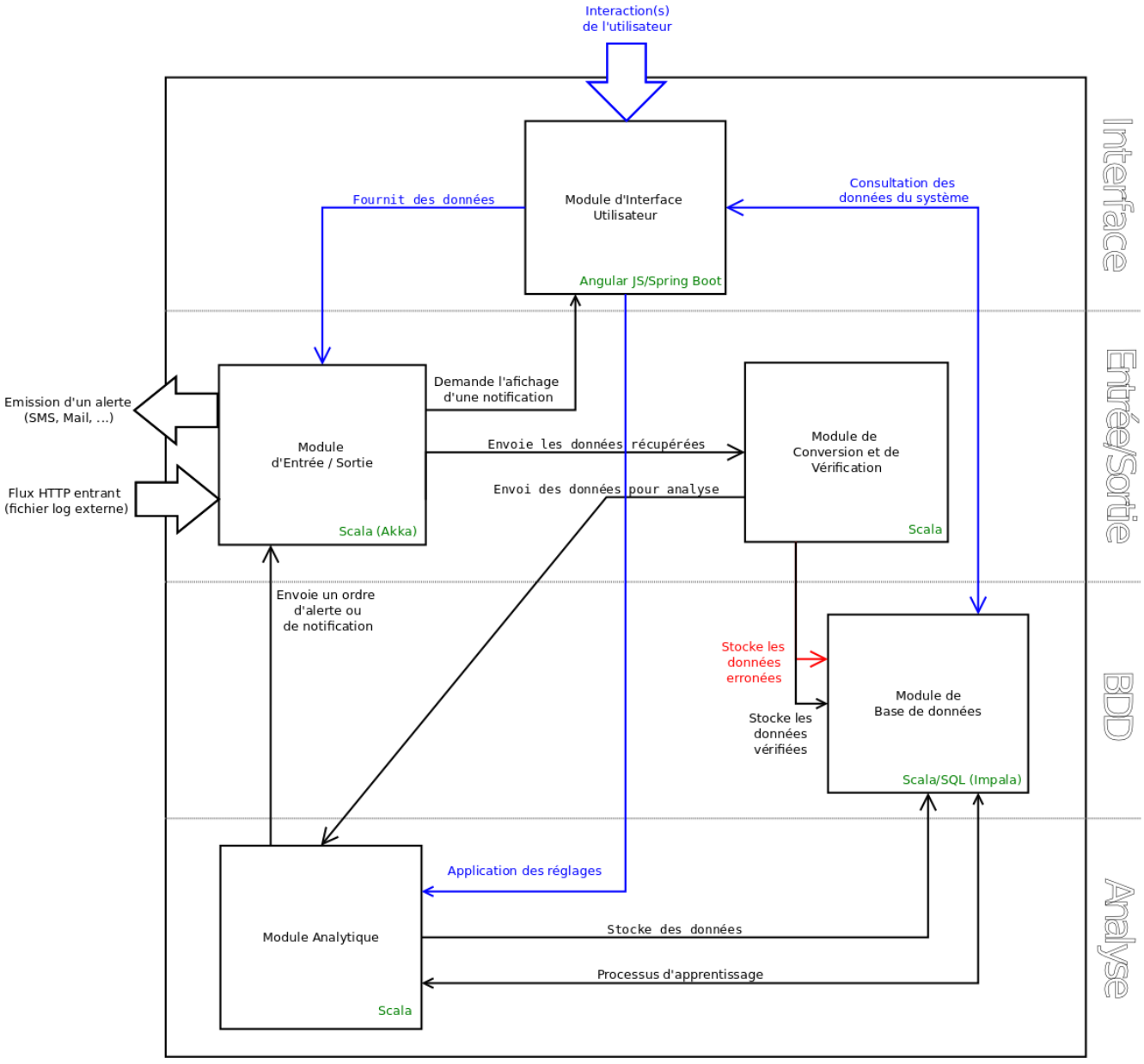
Afin de répondre à un tel besoin, notre client (Mr. Jonathan Germond) nous a fourni une liste de spécifications à suivre, lors de notre premier rendez-vous client :

- Vu que ce projet sera repris en interne par Saagie, il est important que le système soit conçue à la manière d'une API : extensible et facilement maintenable.
- L'analyseur devra être capable de prendre en charge des flux de données en provenance de Shinken, Logstash et Apache Kafka. Ce sont des pipelines, c'est-à-dire, des applications qui ont pour but de collecter et formater des logs.
- Le système devra alerter les opérateurs en employant des moyens de communication utilisés par l'entreprise, tels que l’application Slack, ou l'envoi de mails.
- Le système devra être capable de s'adapter à des flux de données ou des méthodes d’alerte non prédéfinies, qui seront implantées dans le système par de futurs développeurs.
- Le système devra conserver toute donnée ayant été sollicitée par le système, même ces données sont inexploitables.
- Enfin, il devra fonctionner sur de longues périodes de fonctionnement et, dans l'idéal, être opérationnel indéfiniment.

II) Spécificités techniques

1) Architecture globale

Diagramme d'architecture globale



Ce diagramme met alors en évidence la présence de 5 modules majeurs :

- L'Entrée / Sortie : Gestion des interactions du système avec l'extérieur
- L'Interface Utilisateur : Permet à l'utilisateur humain de superviser le comportement su système.
- Le module de Conversion / Vérification : Permet d'assurer l'intégrité des données manipulées
- La Base de Données : Gère la persistance du système
- L'Analyseur : Réalise la fonctionnalité d'analyse et de prédiction

Principe de fonctionnement

A la réception d'une requête HTTP externe par le gestionnaire d'entrée, le système va extraire tous les logs contenus dans la requête et les formater au moyen du module de Conversion/Vérification, dans le but d'uniformiser le traitement.

Ces données sont ensuite stockées dans un tampon (le batch), qui a pour rôle de rendre le travail d'analyse périodique, en définissant des fenêtres d'actions de 10s, et d'éviter une saturation dans le cas d'un afflux trop important de requêtes externes.

Toutes les 10s, ce batch va alors envoyer les logs reçus, à l'analyseur en prenant soin de ne conserver que les logs ayant passé l'étape de vérification (les logs erronés sont alors stockés dans la base de donnée, sans traitement particulier). Une fois l'analyse terminée, les logs se voient attribuer un label de criticité, dont la valeur dépend des informations extraites depuis le contenu de chaque log. Les logs sont alors stockés au sein du lac de données distant.

Une fois le traitement terminé, le système va alors comparer les labels obtenus avec la valeur limite, correspondant à la valeur associée aux défaillances et cas critiques. Le système lancera alors l'alerte grâce à ses mécanismes de levée d'alerte (par Slack et par mail), pour tout log ayant un label supérieur ou égal au niveau de criticité pré-défini.

Mise en place du service

De façon générale, SmartLogger est conçue comme une application RESTFUL, du fait que l'on communique avec ce service via un ensemble de règles présentes dans le cadre d'un service REST.

Ces règles correspondent à la réception des requêtes HTTP suivantes :

Chemin	Méthode	Action
/smartlogger/label/{id}	PUT	Modifie le label associé au log possédant le numéro d'identification fourni en paramètre, dans le chemin.
/smartlogger/train	PUT	Force la phase d'entraînement du système
/smartlogger/analyze	PUT	Demande une analyse sur le jeu de logs reçu
/smartlogger/provide	PUT	Sauvegarde les logs reçus en tant que jeu d'entraînement
(*) /smartlogger/	GET	Affiche l'ensemble des logs contenus dans le système

(*) Disponible sur le service client uniquement

Le but d'un tel service étant d'assurer la communication avec le serveur de requêtes, afin qu'il puisse répondre aux demandes d'analyses, ou pour pouvoir aisément y coupler une interface homme-machine via un service client.

La partie suivante visera à détailler le comportement et le fonctionnement de chacun de ces modules, en spécifiant les technologies mises en place et les algorithmes utilisés.

2) Modules associés au système

Module d'entrée

Le module d'entrée à pour but de recevoir des requêtes HTTP contenant les logs reçus de systèmes externes tels que des serveurs, des applications, ...

L'entrée fonctionne via le framework Akka, permettant la réception de requête HTTP de tout type, afin de permettre la transmission de logs depuis des services externes.

Pour cela, il utilise un *Batch* se vidant toutes les X secondes, afin de stocker les logs reçu depuis l'extérieur, puis d'exécuter une analyse de l'ensemble des logs stocké dans le batch de manière cyclique.

Module de Conversion / Vérification

Ce module permet de "nettoyer" les logs reçus afin de garder uniquement les données valides, dont a besoin SmartLogger pour fonctionner. Il est constitué d'une classe LogParser permettant de parser et de nettoyer les logs reçus afin de ne garder le contenu nécessaire au fonctionnement du module analytique.

Module Analytique

Le module analytique représente le cœur de l'application, celui-ci est constitué de 2 classes.

La première classe du module est la classe SmartAnalyzer, celle-ci a pour rôle d'implémenter les facultés de prédiction et d'analyse de SmartLogger. Utilisant les algorithmes issues du framework Apache Spark et de la bibliothèque ml.classification, cette classe est capable de prédire des événements en fonction d'un modèle prédéfini par l'utilisateur de l'application.

Son rôle dans le projet consiste, à exploiter un ensemble de logs reçus du batch du module d'entrée, pour prédire si ceux-ci sont critiques dans le cadre de sa génération ou bien si ceux-ci sont insignifiants et n'indique aucun problème de fonctionnement du serveur ou du service externe.

La seconde classe, AnalyzerBuilder, permet de pré-construire des analyseurs basés sur la classe SmartAnalyzer, en les définissant à l'aide d'un des types d'algorithmes de classification fourni par l'API Spark. Elle permet ainsi l'exploitation des algorithmes suivants :

- NaiveBayes
- LogisticRegression
- DecisionTreeClassifier
- RandomForestClassifier

Module de Base de Données

Le module de base de données est le module gérant la persistance des données au sein de SmartLogger.

Pour cela, il est constitué d'un ensemble de classes ayant pour but de créer et de manipuler une base de données Impala, afin notamment de pouvoir stocker l'ensemble des logs reçus par SmartLogger. Il définit donc un niveau d'abstraction supplémentaire permettant de lier un type de données manipulé par le système (type Scala correspondant au log) à une ligne de table donnée (ici, une ligne de table de données représentant un log).

Pour cela, ce module implémente un pattern DAO, qui exploite des connecteurs (instances de la classe DBConnector) permettant la connexion à la base de données via les éléments de connexions fourni par le fichier de configuration, ainsi qu'une seconde classe permettant la manipulation courante (cd : Méthode CRUD) des éléments dans la base de données.

Module d'Interface Utilisateur

Ce module est externe au projet SmartLogger. En effet, cette interface est facilement interchangeable par une autre interface, tant que les règles du service de SmartLogger sont respectées.

Ici, on y retrouve les trois frameworks suivants : Angular JS, Bootstrap Twitter et Spring Boot. Les deux premiers utilisés dans le cadre de la Vue de l'interface. Le troisième est quand à lui utilisé côté Back-End, afin de définir les routes permettant de communiquer avec SmartLogger.

Module de Sortie

Le module de sortie de SmartLogger permet, lors de la détection d'un niveau de criticité important, de prévenir les équipes via un envoi d'e-mail contenant les logs critiques ou via le logiciel de discussion instantané Slack.

Pour cela, SmartLogger possède deux interfaces Alerter et Notifier, qui permettent d'établir un patron Observateur, dans le but de faciliter la façon dont les personnes concernées seront contactées. L'alerteur représente la levée d'alerte alors que les notifieurs représente le moyen de communication, tel qu'implantés par ses classes correspondantes.

Nous retrouvons ici l'API Slack pour l'envoi sur une sortie Slack et la librairie Javax.mail pour l'envoi de mail depuis une classe Scala.

Module utilitaire

A ces différents modules, nous retrouvons un module complémentaire en la présence de classe utilitaire.

Ces classes sont utilisées afin d'externaliser un ensemble de configuration hors du code dans des fichiers properties.

Ces classes permettent ainsi de charger des fichiers properties afin d'intégrer au code les clés d'API de Slack, les adresses mails, ou encore d'autres configurations qu'il est nécessaire d'externaliser, afin notamment de faciliter le déploiement de l'application sur un serveur prédéfini.

3) Outils et Technologies employées

Maintenant que nous avons défini ensemble les différentes modules et les technologies liées, nous allons revenir sur les technologies de façon plus détaillée, afin d'en montrer l'intérêt dans le cadre de SmartLogger, tout en spécifiant les technologies potentielles qui auraient pu être utilisées, en lieu et place de ce qui est présent actuellement.

Un langage pour tout unifier : Scala

Scala est une surcouche du langage de développement Java. Scala est un langage multi-paradigme permettant le développement via le paradigme objet ainsi que fonctionnelle. Nous avons choisi d'utiliser Scala dans l'entièreté du projet, car celui-ci procure un ensemble d'avantages non négligeables.

Spark étant principalement rédigé en Scala, nous avons souhaité poursuivre entièrement en Scala, car celui-ci présente quelques avantages notamment le fait qu'il est moins verbeux que le Java, allégant ainsi le code produit.

De plus, il est plus performant que le Java lors d'opération multi_thread, chose essentielle pour le module Analytique de SmartLogger. Pour finir, sur un plan purement didactique, la manipulation de ce nouveau langage nous a permis de nous confronter à de nouveaux modules de développement, ce qui accentue encore l'utilité du Scala dans le projet.

Un framework de gestion des flux HTTP : Akka

Afin de suivre la continuité de développement en Scala, nous avons opté pour le framework Akka afin de gérer les requêtes HTTP reçus depuis les services et systèmes externes.

En effet, en plus de répondre au besoin que nous avons, celui-ci se révèle apte à supporter un ensemble de requêtes bien plus important que ce que nous lui imposons pour le projet SmartLogger. Concrètement, en cas de passage a une échelle supérieur de l'application, avec la réception de plusieurs milliers de logs à la seconde. Nous savons que SmartLogger pourra supporter la charge sans devoir revoir le fonctionnement du module d'entrée.

Les flux HTTP auraient pu être gérés via le framework Spring-Boot, notamment via le principe de contrôleurs REST, permettant, en fonction d'une URL précise (une route), de définir des actions à effectuer sur SmartLogger, tel que la récupérer de l'ensemble des logs stockés dans la base de données, ou encore le ré-entraînement du modèle afin de le corriger.

Cependant, dans l'optique que nous avions défini, nous avons préféré utiliser Akka, car celui-ci permettait de faire évoluer les futurs potentiels besoins de SmartLogger sur de gros volume de logs à traiter.

Un framework pour la gestion de l'apprentissage : Apache Spark

Afin de gérer l'apprentissage automatique de l'application, notre client nous a suggéré de nous axé vers la bibliothèque mlLib.classification du Framework Apache Spark, ce que nous avons fait dans un premier temps, puis après une mise à jour importante du Framework, l'attention s'est porté sur la bibliothèque ml.classification, celle-ci étant dédié au nouveau module de Spark, géré par sa nouvelle machinerie nommé SqlSession.

Après des essais, il était assez clair que l'un ou l'autre était parfaitement adapté à notre besoin, le changement n'apportant simplement que des optimisations temporels et visuels dans le code.

Une librairie pour l'API Slack : Scala-Slack

Concernant le contact auprès de l'API Slack, il était nécessaire de faire des recherches auprès du site de celui-ci. Slack étant utilisé par de plus en plus de développeurs, nous avons pu utiliser une bibliothèque dédié à la communication entre une application et l'API Slack.

Un gestionnaire de dépendance : Gradle

Afin de gérer les dépendances du projet, nous avons choisi d'utiliser Gradle. En effet, Gradle est un gestionnaire de dépendance utilisant le langage Groovy afin de gérer le téléchargement et l'utilisation des dépendances du projet SmartLogger.

Nous avons choisi d'utiliser Gradle car celui-ci est le gestionnaire le plus moderne, et celui qui semblait être le plus prometteur. Cependant, sa modernité rends la documentation de l'outil difficilement maniable due à une faible utilisation en comparaison avec des outils plus anciens tels que Ant ou encore Maven.

De ce fait, nous aurions pu utiliser Maven comme gestionnaire de dépendance, mais au vu des technologies que nous utilisions, nous avons opté pour les technologies les plus récentes possible, nous permettant par ce biais de nous former et nous intéresser à des technologies nouvelles.

III) Assurer la qualité

1) Démarche et stratégies appliquées

Une application de la taille de SmartLogger se doit d'être une application de qualité. Grâce à une certaine gestion de la qualité, nous avons pu assurer la robustesse, la maintenabilité, la conformité et la fiabilité de l'application, cela à moindre coût.

Pour cela, l'ensemble de mesures ont été prises, seront détaillées dans les sections suivantes. Notons que la démarche du test-driven developpement a été adoptée puis abandonnée au début du développement, car la nature des technologies ainsi que notre faible expérience dans la gestion d'un projet de cette taille, aurait rendu cette méthode inefficace voire contre-productive.

Tests unitaires

Pour chaque classe de l'application, nous avons effectué des tests élémentaires, afin de vérifier le bon fonctionnement de l'application, sa capacité à réagir en cas de mauvaise entrée, ainsi qu'en son comportement face d'erreurs.

Les tests unitaires représentent le plus gros domaine de la volumétrie de test, et au final, ceux-ci ne testent qu'une fonctionnalité chacun, pour un module donné, à un temps donné. Celai ne les empêche pas d'être cruciaux et vitaux au bon fonctionnement de l'application.

Pour effectuer les tests unitaires, nous avons majoritairement utilisé ScalaTest, ce dernier étant le framework qui nous a semblé le plus adapté, étant donné notre environnement de développement ainsi que le langage sur lequel est basé l'application, en grande partie en Scala.

Nous avons aussi utilisé Mockito-Sugar, afin de tester certaines classes qui n'étaient pas testables seules, à cause de contraintes liées à des dépendances envers d'autres classes, soit non développées, soit trop lourdes à instancier. Ainsi, ces dernières sont mockées afin de simuler leur comportement, sans avoir à les instancier.

Enfin, il est important de noter que la condition de validité d'une classe, est le passage de 90% de ces tests, ainsi que le passage de 100% des tests critiques, tests étant défini comme déclencheur de défaillance lors d'une exécution de l'application.

Intégration

Après avoir effectué les tests unitaires de différents modules, des tests d'intégration ont été effectués. L'objectif de ces tests est de contrôler les communications entre plusieurs modules, testant ainsi leur inter-opérabilité. Leur but final est de tester l'architecture globale de l'application pour une itération donnée.

Montée en charge

Certains modules ont été testés en situation de stress, c'est-à-dire en observant leurs comportements en cas de sollicitation élevée du système. Cela s'applique notamment aux modules d'entrée et de sortie, ainsi qu'au le module analytique, vu qu'ils sont les plus exposés à ce type de problème.

Dans cette optique, nous avons eu recours à Gatling, qui est un framework dédié au test de performance et de réaction. Cela nous a permis de simuler l'envoi d'une quantité massive de log, par requête HTTP, et de suivre la capacité du système à les recevoir.

Performances

Afin d'évaluer la performance du module analytique, nous avons effectué du parangonnage, par lequel nous avons pu comparer la précision, l'efficacité et le coût des différents algorithmes de classification. Tout cela conformément aux exigences client.

Grâce à ces tests, nous avons pu mettre en valeur l'efficacité de deux algorithmes en particulier : NaiveBayes et DecisionTreeClassifier.

Comportementaux

L'IHM a été, quand à elle, testée visuellement afin de vérifier que son rendu soit conforme à celui spécifié dans les documents de référence.

L'IHM étant un module de moindre importance, elle n'a été testée que sommairement. Mais différent frameworks ont été recherchés pour l'occasion, et si l'on devait étendre l'IHM, nous utiliserions Cuncumber.

Validation client

Un ensemble de réunions avec le client ont été mises en place, afin qu'il puisse suivre l'avancement de l'application et valider ses avancées, ainsi que suggérer de nouveaux ou différents axes pour les modules.

Pour cela, des simulations de fonctionnement de l'application, des démonstrations ont été réalisées, et soumises au client.

2) Evolution de la qualité au cours du développement

Assurer l'évolution de la qualité

Afin d'assurer que le code fonctionne comme prévu, il est nécessaire de le vérifier à l'aide de tests. Ces tests ont plusieurs buts tels que : vérifier la fonctionnalité du code (qu'il produit la sortie attendu), est capable de s'adapter aux différentes situations et erreurs possibles (assurer la robustesse du code) et qu'il s'intègre parfaitement au reste de l'application.

Lorsqu'on prévoit les tests, il faut assurer que le code soit testé sur le plus d'aspects possible, c'est le principe de la couverture du code. Le but des testeurs est donc de s'approcher le plus possible des 100 % de couverture de code.

Sur ce type de projet, il est cependant impossible d'obtenir 100 % de couverture sur le code, du fait qu'il n'est pas possible de prévoir absolument toutes les entrées possibles et imaginable pour un code. Nous avons donc procédé en faisant tendre ce pourcentage de couverture vers le maximum possible. Ainsi, lorsque nous parlons de 100 % de couverture, ce sera le 'presque 100 %' que nous avons explicité plus haut.

Pour être capable d'obtenir une couverture de 100 %, nous nous sommes basés sur le cahier de recette, qui liste précisément l'ensemble des tests à produire, afin de s'assurer de ne rien manquer. De plus, il est important de bien structurer les tests sous forme de scénario, afin de faciliter la maintenance et l'efficacité des ces tests.

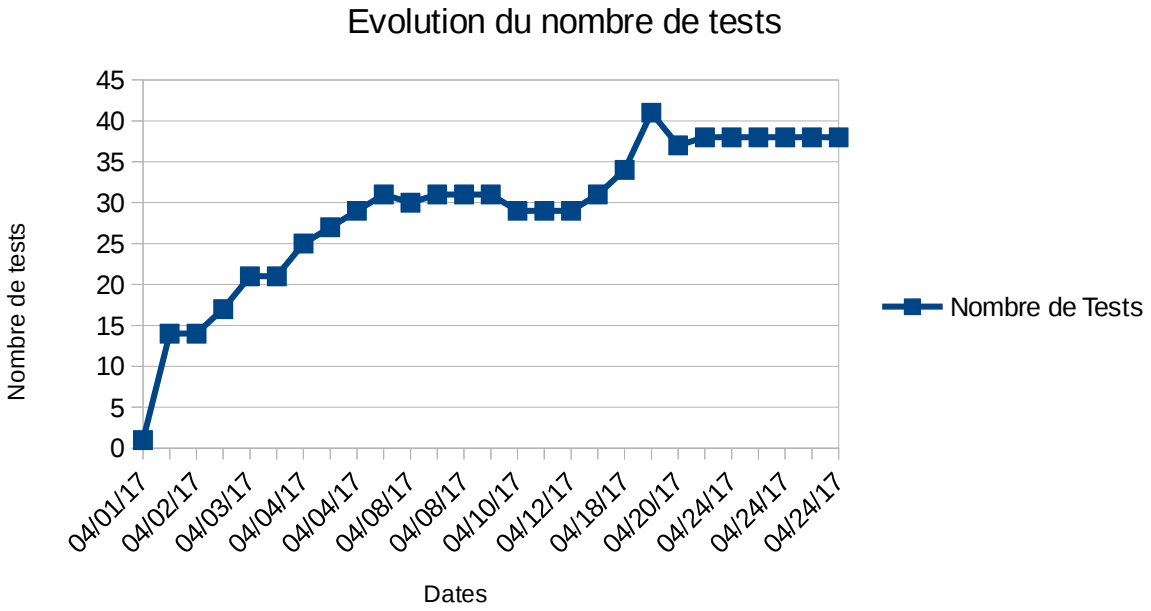
Il nous est alors possible de produire chaque test et vérifier que le code testé produit les résultats attendus.

Il est intéressant de rappeler qu'il est néanmoins possible que le code testé peut lui aussi changer. A ce moment là, il est possible de devoir ajouter, modifier ou supprimer des tests. Dans les cas les plus extrêmes il peut être nécessaire de devoir entièrement supprimer une classe de tests ou faire une restructuration complète. Ces deux cas sont déjà survenus au cours du projet, et ont pu être solutionnés par la méthode que nous avons appliquée.

Couverture sur la 3è itération

La troisième itération a été la plus marquante en termes de stratégie qualité. Le processus d'intégration nous a poussé à adopter un tracking plus soutenu des activités de tests, dans le but de vérifier la qualité de notre stratégie vis-à-vis des gros changements pouvant être instaurés par le processus d'intégration.

Le graphe suivant décrit l'évolution du nombre de tests applicable, sur la période de la 3ème itération.

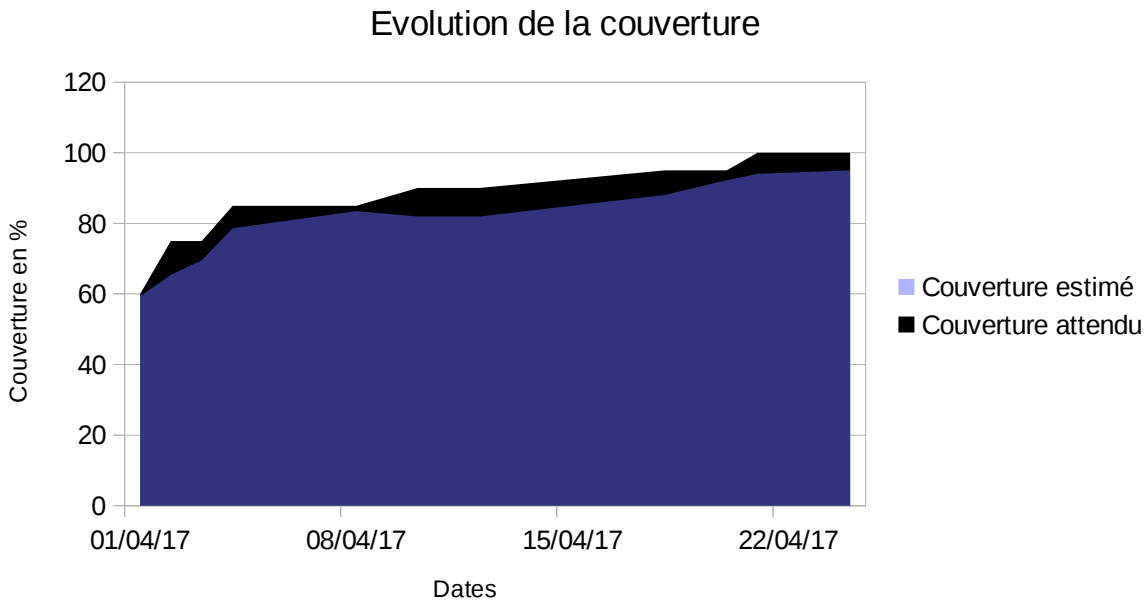


Notons que certains tests n'ont pas été inclus dans ces chiffres (par cause de redondance, ou de pertinence).

On peut constater que sur la période de début d'intégration (du 01/04/17 jusqu'au 08/04/17), nous pouvons voir une augmentation constante du nombre de tests. Cette période correspond à l'écriture de la plupart des cas de tests.

La seconde période (du 04/08/17 jusqu'au 18/04/17), montre une stagnation du nombre de tests. Ceci correspond à la correction des tests afin d'assurer que les tests déjà écrits passent. Finalement la dernière portion correspond à la mise-à-jour des tests suite à une mise-à-jour du code.

Ceci fait, nous pouvons regarder le graphe représentant l'évolution de la couverture sur la 3ème itération.



Nous observons donc que l'écriture des tests fut important : le 01/04/17, seul 60 % de taux de couverture était en place : (donc 40 % des cas possibles n'étaient pas pris en compte).

Nous observons aussi que, même si nous n'avons pas réussi à atteindre le niveau de couverture attendu, nous avons néanmoins réussi à atteindre un taux de couverture largement suffisant (95%) pour pouvoir témoigner de la qualité des fonctionnalités produites.

En terme de couverture à la fin du développement, nous obtenons les indicateurs suivants, qui nous confortent dans l'idée d'avoir réalisé des fonctionnalités de qualité.

Module	Couverture	Taux de passage	Acceptation
Analytique	100 %	100 %	
Entrée	100 %	100 %	
Sortie	100 %	100 %	
Utilitaire	100 %	100 %	
Persistance	90 %	90 %	
Interface	90 %	100 %	

IV) Rétrospective du développement

Le contenu des sous-parties suivantes vont se présenter sous la forme de points-clés, qui ont été fournis par tous les membres de l'équipe, vis-à-vis des différents sujets abordés.

1) Apports du projet

Points positifs soulevés par l'équipe

- Amélioration des compétences liées à la programmation
- Familiarisation et apprentissage de l'utilisation des frameworks
- Valorisation du travail, du fait qu'il correspond à un cas métier concret.
- Découverte des procédés de mise en place de stratégie qualité
- Mieux appréhender ce qu'est une équipe et la façon de la gérer
- Comprendre et appréhender les méthodes AGILE
- Travailler pour un intervenant extérieur

Synthèse

La plupart des points positifs soulevés par l'équipe, sont liés au contexte de réalisation du projet. Le fait que le projet vienne d'une entreprise a su motiver plus que d'ordinaire, et nous a permis d'appréhender des technologies, dont nous n'avions que des notions, tout au plus.

Les intervenants extérieurs que nous avons rencontrés au cours du projet, ont pu nous fournir des critiques constructives ainsi que de nombreux conseils pour nous guider :

- A la spécification, pour nous indiquer quelles technologies utiliser.
- Au développement, pour mettre en avant des fonctionnalités auxquelles penser, aux points à mettre en avant, etc.

Dans l'ensemble, il a également permis à chacun de se forger une opinion sur le travail en équipe de par la différence des profils et l'implication de chacun.

2) Problèmes et difficultés rencontrées

Problèmes rencontrés selon les membres de l'équipe

- Problèmes d'organisation liés à la clarté de la répartition des tâches
- Trop de technologies à assimiler : bon niveau mais pas de réelle maîtrise
- Manque de communication claire entre les membres de l'équipe
- Manque d'implication de certains membres de l'équipe sur certaines périodes du projet
- Gestion de projet trop laxiste sur les retards de ses membres.
- Complexification inutile sur certains aspects du projet
- Les cours associés aux connaissances utiles sont intervenus trop tard
- Inefficacité de certaines pièces documentaires
- Majeure partie du développement à 5

Synthèse

Il y a trois facteurs principaux qui expliquent les différents points cités ci-dessus.

Le premier est le manque d'expérience vis-à-vis des technologies à manipuler, ce qui a engendré des temps d'apprentissage supplémentaires très élevés (entre 20 et 40 % du temps total). Ce style d'apprentissage est également lié à l'autonomie de l'équipier : Plus un élément est autonome, plus il a la facilité d'apprendre facilement et de transmettre son savoir à l'équipe, ce qui nous amène au second point.

Le point suivant correspond au manque d'organisation de l'équipe en règle générale, et ce, malgré l'instauration de plusieurs moyens de communication, de rendez-vous journaliers, de dialogues entre les cours. Le fait est, que les profils des acteurs de l'équipe sont vraiment différents tout comme la vision de la gestion de projet idéale : Là, ou certains aurait prêté une attention toute particulière aux retards tout en conservant une bonne autonomie, d'autres auraient préférés être guidés tout du long, en ayant des descriptifs plus précis sur ce qu'il y avait à faire. Ainsi, il y a eu un problème d'efficacité en termes d'adaptation pour une organisation qui se voulait être un entre-deux entre autonomie et pilotage. Notons toutefois que ces difficultés nous ont empêchés d'être une organisation optimale, mais pas d'être une organisation qui fonctionne puisque nous avons tout de même réalisé les grandes lignes de ce projet.

Enfin, l'organisation temporelle des cours et la démarche imposée. Cette année, les savoirs que nous avons acquis au cours du second semestre, aurait pu intégralement changer la façon dont nous aurions pensé l'architecture du projet. Par ailleurs, certaines pièces documentaires ont nécessités de nombreuses heures sans utilité particulière : Là, où la STB et la majeure partie de la DAL ont pu nous guider tout au long du projet, l'architecture technique du DAL, le PdD et l'AdR n'étaient pas d'une grande utilité, du fait que nous n'étions pas enclins à fournir de telles informations à ce moment.

3) Axes d'amélioration possibles

Sur le plan humain, l'amélioration organisationnelle principale est l'instauration d'une organisation plus formelle avec des points de contrôle plus adaptés (rendez-vous hebdomadaires, par exemple). Avec une expérience plus élevée, le TDD (Test-Driven Development) aurait également pu être une organisation plus adaptée à ce type de projet.

Sur le plan technique, l'amélioration principale concerne le passage de l'application en utilisant le framework Spring. La mise en place d'Akka permet de mieux gérer les données en terme de volume, mais Spring offre plus de fonctionnalités tel que la gestion simplifiée d'un service REST, des fonctionnalités de persistance, ce qui nous aurait permis d'économiser le temps de réalisation d'un module entier. En bref, opter pour l'emploi de solution existantes (frameworks, toolkits, ...) au lieu de se baser sur des implémentations d'équipe.