École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

Département Informatique de Polytech Tours

Rapport de stage 2019 - 2020

Docker en action

Entreprise
Worldline
80 Quai Voltaire
95870, River Ouest

Tuteur.rice entreprise
Basile Gathignol
[basile.gathignol@worldline.com]
Responsable d'équipe

Étudiant.e
Colin Troisemaine
[colin.troisemaine@gmail.coml]

Tuteur.rice académique
Gilles Venturini
[gilles.venturini@univ-tours.fr]

Version du 23 août 2020

Table des matières

Table des figures

Chapitre 1

Remerciements

Avant de démarrer ce rapport de stage, je tiens à commencer par remercier les personnes qui m'ont permis de réaliser ce stage et ont eu la gentillesse de me faire profiter de leurs connaissances. Je tiens à remercier :

- Basile GATHIGNOL qui m'a offert l'opportunité de cette expérience professionnelle et m'a accordé sa confiance et confié des missions que j'espère avoir bien accomplies.
- Kévin TOUBLANC pour le temps qu'il m'a consacré et pour m'avoir guidé tout au long du stage avec une grande pédagogie et bienveillance.
- Anaëlle HAMON qui m'a accordé beaucoup d'attention et a répondu mes interrogations d'étudiant.
- Jihad OUSSAD qui s'est rendu disponible pour me donner tous les conseils dont j'ai eu besoin.

Chapitre 2

Introduction

Ce stage que j'ai effectué pendant les mois de juin, juillet et août 2020 s'est déroulé au sein de l'entrerpise Worldline, sur son site de Tours, en Indre-et-Loire. J'étais plus précisement intégré à l'équipe "Sips" qui travaille sur une application proposant des solutions de paiement en ligne. Celle-ci est actuellement adoptée par un grand nombre de sites marchands et représente une grande part des transactions liées aux achats en ligne français.

Au cours de ce stage, j'ai eu l'occasion de travailler sur deux sujets principaux qui portaient sur deux applications du même système, ce qui m'a permis de me familiariser avec le fonctionnement général du système de la solution Sips. J'ai donc d'abord travaillé à mettre en place une génération de logs plus facilement interprétables afin de rendre la recherche d'erreurs plus simples, puis j'ai participé au processus de Dockerisation d'une application et à la rédaction de tests d'intégration sur ce même projet.

Dans ce rapport, je vais commencer par présenter l'entreprise Worldline et plus particulièrement le produit géré par l'équipe que j'ai intégrée. Puis pour les deux projets auxquels j'ai participé, je présenterai le besoin lié au sujet, puis la solution qui lui a été apportée, tout en détaillant les différentes technologies utilisées.

Chapitre 3

Présentation de l'entreprise

3.1 Worldline en quelques mots

Fondée initialement par la fusion des deux entreprises Axim et Sligos en 1973, Worldline est une société disposant de presque 50 ans d'expertise dans l'industrie du paiement. Aujourd'hui, elle est constituée de plus de 10 000 collaborateurs répartis dans 17 pays et possède un chiffre d'affaire de 2,3 milliards d'euros en 2019.

Worldline est divisée en 3 *Business Units* qui représentent ses activités principales :

— **Mobility & e-Transactional Services** (MTS) vise à offrir aux consommateurs une expérience différente via les nouvelles technologies au travers des applications connectées. Elle a su gagner de nouveux clients par différentes solutions comme par exemple l'e-education de la solution « SQOOL » qui offre une nouvelle méthode d'apprentissage dans les écoles en utilisant des tablettes spécialement conçues à cet effet. Cette solution a été testée en 2015/2016 avec succès au sein de 500 écoles et a été utilisée par 70 000 écoliers.
Worldline a également collaboré avec Michelin pour développer un nouvel outil de gestion de remorques en temps réel et avec Bouygues pour permettre des achats en ligne à l'aide du e-ticketing.

FIGURE 3.1 – Clients de la Business Unit MTS

— **Financial Processing & Software Licensing** (FPL) a principalement pour but de gérer les flux de transactions entre acquéreurs et émetteurs (c'est à dire entre la banque du client et la banque du commerçant). Voici quelques exemples de projets qu'elle a menés : Une dématérialisation d'une partie de l'administration publique de l'AFL, une solution de paiement par NFC via carte SIM pour Postbank ou encore un système d'authentification facile sur des sites de e-commerce avec la banque EastWest.

FIGURE 3.2 – Clients de la Business Unit FPL

— **Merchant Services & Terminals** (MS&T) a pour objectif d'améliorer les services des marchands dans le but de proposer une expérience nouvelle au consommateur. Cette Business Unit a par exemple réalisé avec Chiltern Railways une solution de billeterie interactive

sur des bornes que l'on peut trouver à Oxford, Parkway et Marylebone. On peut également trouver des bornes d'achat de sa conception chez Sephora ou une solution de paiement par QR code pour régler des rechargements de véhicules électriques chez Powerdale.

Figure 3.3 – Clients de la Business Unit MS&T

Ainsi, Worldline intervient dans toutes les étapes de la chaîne de paiement de ces projets par le biais des métiers liés à ces Business Units : Transport et gouvernement (MTS), acquisition (FPL) et acceptance (MS&T).

Figure 3.4 – Worldline dans la chaîne de paiement

3.2 Organisation de l'entreprise

3.2.1 Organisation générale

Parallèlement aux 3 Business Units qui représentent ses activités, Worldline peut être actuellement représentée selon 4 groupes principaux qui vont rassembler les différentes équipes dont fait partie Sips que j'ai intégrée durant mon stage :

- **Customer Support & Architecture** (CSA) : Comme son nom l'indique, ce groupe est principalement chargé de réaliser le suivi client et production et il possède également une équipe d'architectes techniques.
- **GTDC** : Composé de deux équipes, ce groupe réalise des tests métiers de non régression et assure le bon déroulement des releases et des projets dans le respect des contraintes de délai et qualité.
- **Produit** : Ce groupe a pour fonction d'identifier les potentiels clients et leurs activités et de définir les différents buts à atteindre.

— **Project Center** : On peut y trouver 3 équipes. Une équipe d'architectes logiciel, une équipe "Extranet" qui assure le développement d'outils de suivi destinés aux commerçants et enfin l'équipe Sips-Core liée au traitement des paiements et au contact avec les banques.

3.2.2 L'équipe Sips

La solution Sips (Secure Internet Payment Services) fait partie de la Business Unit MS&T et plus particulièrement de l'équipe Project Center. Elle est responsable du développement des outils qui permettent aux commerçants de proposer à leurs clients des paiements à distance et de pouvoir les gérer par la suite (rembourser, différer, surveiller, ...), cela sans qu'ils n'aient besoin de stocker des données sensibles (comme des numéros de cartes, des données personnelles, ...). Comme Sips a besoin de conserver ces données, elle possède la certification PCI-DSS (*Payment Card Industry - Data Security Standard*) qui a été mise en place par VISA et MASTERCARD et a pour but renforcer la sécurité des données des utilisateurs et de faciliter l'adoption de mesures de sécurité uniformes à l'échelle mondiale.

On distingue quatre interfaces au sein de la solution Sips qui possèdent toutes un rôle précis :

— **Sips Paypage** regroupe un ensemble de pages internet permettant de réaliser des transactions de bout en bout.
— **Sips Office** permet de réaliser des transactions en Machine To Machine (M2M), ainsi le commerçant peut intégrer les différentes fonctionnalités de Sips au sein de ses propres pages de paiement.
— **Sips Office Extranet** offre aux commerçants la possibilité de gérer des transactions effectuées (remboursement, annulation, validation, duplication, ...).
— **Sips Office Batch** permet de réaliser les mêmes opérations que Sips Office Extranet, mais en masse par le biais de batchs.

En plus de ces interfaces, un système de reporting via des journaux permet d'informer les marchands des mouvements financiers réalisés au travers de son site web (transactions, opérations, paiements, impayés...). Les clients de Sips ont ainsi la possibilité de choisir quel reporting ils souhaitent pour le suivi de leur activité.

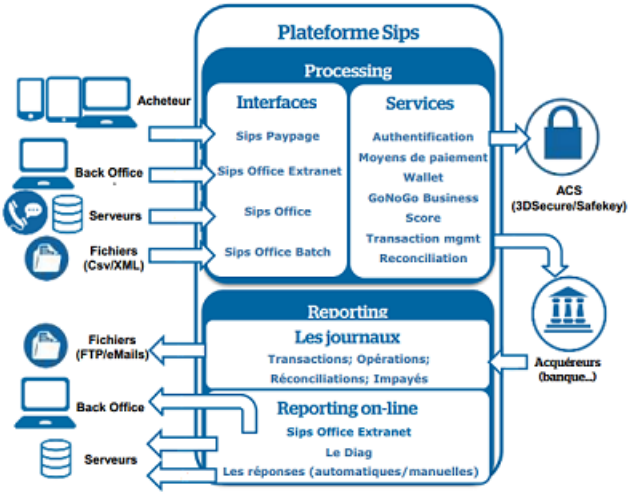

FIGURE 3.5 – Architecture fonctionnelle de Sips

Chapitre 4

Premier sujet : Logs en info

4.1 Présentation du besoin

Lors d'incidents en production, il est utile de connaître les informations envoyées vers les acquéreurs (≈ les banques qui reçoivent la transaction). Toutes ces transactions vont actuellement transiter par le module *sips-cars* qui centralise les connexions et permet une éventuelle simulation en fonction des options du commerçant, c'est donc lui qui sera chargé de stocker les logs générés. La plupart de ces connexions sont dirigées vers le module SARA (qui est chargé d'envoyer les transactions vers les bons acquéreurs) et se font en NLV (Numéro Longueur Valeur) qui est un mode de stockage de données propre à Worldline. Les différents modules PM (Payment Means) envoient leurs buffers NLV au travers du module CARS qui va s'occuper de les rediriger et de leur renvoyer la réponse du module SARA.
Ainsi, lors d'accidents en production, SARA est en mesure de connaître le contenu de certaines trames, mais il est actuellement difficile de lier cette trame à une transaction de Sips et donc de retracer l'origine des erreurs.

4.2 Présentation de la solution

Plusieurs mécanismes devront être mis en place, afin de répondre à cette problématique :

- Identifier le message envoyé vers SARA afin de rapprocher les transactions facilement.
- Afficher les logs de niveau INFO sur les environnements de production.
- Permettre de rendre les logs plus lisibles (associer un numéro à un nom métier et les formater en JSON pour faciliter leur lecture).
- Intégrer la trace dans le système de monitoring (Kibana).

Il existe trois types de buffers NLV que le module CARS reçoit et renvoie (chaque buffer est associé à un numéro que l'on peut utiliser pour faire le lien avec un nom métier comme "numéro de carte", "quantité", "nom du marchand", ...) :

- Valeur : Numéro / Longueur / Valeur.
- Liste : Numéro / Longueur / Tableau de numéros + buffers NLV.
- Buffer : Numéro / Longueur / Buffer NLV.

Le module CARS devra donc être capable de les lire et de les analyser afin de générer les logs attendus.

Les contraintes PCI et GDPR impliquent que certaines données ne doivent pas apparaître dans les logs. C'est notamment le cas pour :

- Le numéro de carte qui doit être partiellement masqué (6 premiers et 4 derniers chiffres en clair, par ex : 497886#########1234) (PCI-DSS).
- Le CVV (cryptogramme) qui doit être totalement masqué (par exemple le CVV 123 devient ###) (PCI-DSS).
- Les adresses IP, emails, données personnelles (adresses, etc) doivent être totalement mas-

quées (GDPR).
Les numéros NLV utilisés pour les données personnelles et les données de carte peuvent changer en fonction du protocole utilisé (même si les numéro NLV sont généralement utilisés pour les mêmes données (par ex : 58 = numéro de carte)). Ainsi, Sips-cars devra être adaptable par protocole pour définir quelles données doivent être masquées.

Puisque seuls les modules PM sont en mesure de correctement associer les numéros des buffers NLV aux différents noms de protocoles (puisqu'un même numéro peut désigner deux protocoles différents pour deux moyens de paiement), il va falloir créer une nouvelle méthode d'appel du module CARS qui va prendre en paramètre, en plus des attributs habituels (le nom du service et le buffer NLV par exemple), un objet permettant de faire le lien entre les numéros NLV et le nom des protocoles, ainsi qu'une manière de savoir comment masquer la valeur.

Il est possible de répondre à cette problématique de deux manières différentes :

- La première serait une nouvelle classe « ProtocolConfiguration » dans laquelle on ajouterait autant de lignes que nécessaire comportant les informations suivantes : numéro NLV, nom de protocole et type de masquage à appliquer au moment du log. Ce nouvel objet serait ensuite passé à CARS en paramètre d'une surcharge de la méthode actuelle utilisée pour communiquer (invoke).
- La seconde est assez similaire et consisterait à créer un nouveau modèle d'objet NLV qui comporterait directement le nom du champ, le numéro NLV, la valeur NLV et la méthode masquage. La particularité de cette méthode est l'introduction du builder design pattern pour créer un nouvel objet CarsService. En effet cela permettrait d'éviter de créer des objets Invoker avec trop de paramètres de construction et rendrait plus claire leur création (avec en contrepartie plus de code à écrire du côté du module CARS). Enfin il suffirait d'appeler objet CarsService en lui passant notre nouveau modèle de NLV pour communiquer avec SARA.

Je n'ai eu le temps de mettre en place complètement que la première méthode avant de passer sur le second sujet de mon stage.

4.3 Le design pattern « builder »

Afin de conserver une rétrocompatibilité maximale, il a été décidé de ne pas modifier les paramètres des méthodes appelées de ce module utilisées dans les autres modules. Ainsi pour fournir la configuration des numéros NLV au module CARS chargé de produire les logs, j'ai créé un nouveau constructeur prenant cette configuration en paramètre. De cette manière il est possible d'utiliser les anciennes méthodes qui se serviront de l'ancien logger et les nouvelles méthodes avec le nouveau logger.
Cette création d'un nouveau type de logger a été l'occasion d'introduire le patron de conception (ou design pattern) monteur (ou builder). En effet pour éviter d'avoir à réaliser le choix du type de logger à utiliser au sein de la classe instanciant le logger, on préfère utiliser une classe intermédiaire qui va renvoyer un nouveau logger approprié.
Voici le diagramme UML :

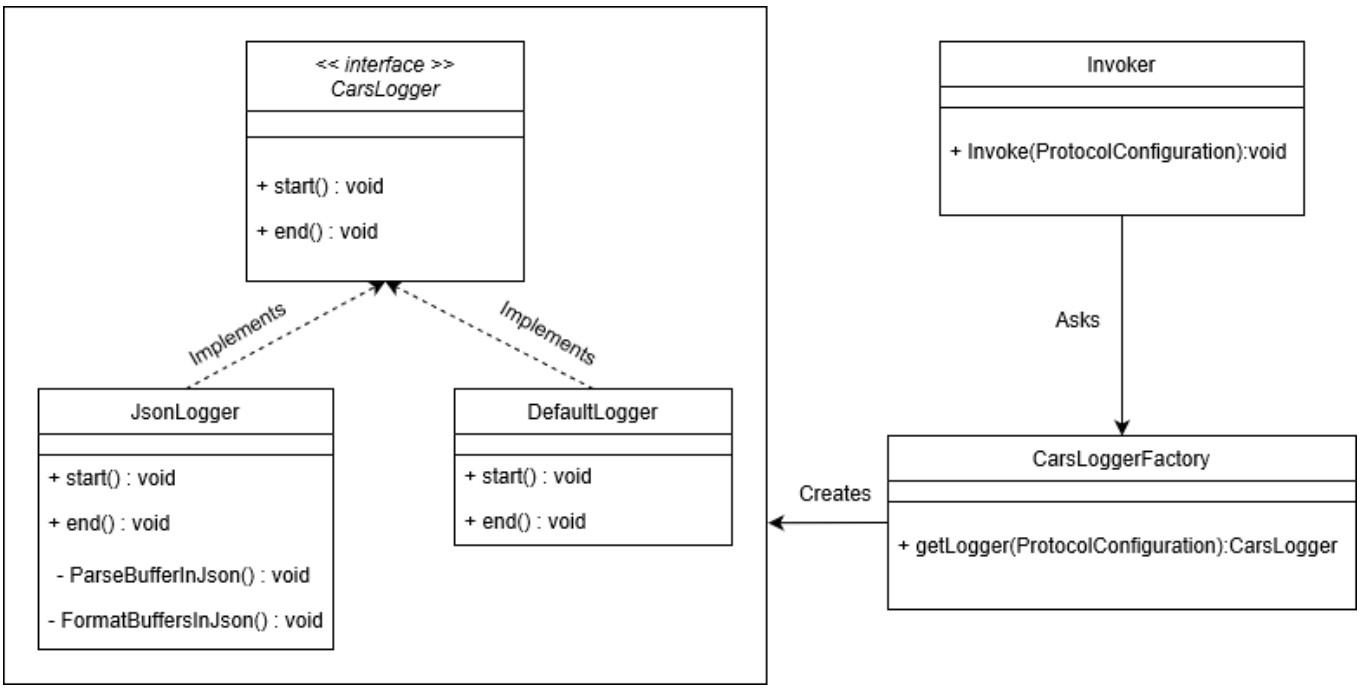

FIGURE 4.1 – Diagramme de classes - Builder design pattern

La classe principale Invoker va donc appeler CarsLoggerFactory en lui donnant la configuration (donc null si aucune). Ainsi c'est cette classe intermédiaire qui va s'occuper de décider s'il faut instancier un JsonLogger (si la configuration est non nulle) ou un DefaultLogger (si elle est nulle).

4.4 Réalisation de tests

4.4.1 Quelques règles de rédaction des tests

Afin de faciliter la relecture et la clarté des tests rédigés, j'ai suivi quelques règles de rédaction et utilisé différentes technologies :

- La bibliothèque Java **AssertJ** fournit une interface fluide pour écrire des assertions. Son objectif principal est d'améliorer la lisibilité du code de test et de faciliter la maintenance des tests. Je me suis notamment servi de la fonctionnalité *SoftAssertions* qui permet de recueillir toutes les erreurs d'assertion au lieu de s'arrêter à la première.
- Une autre fonctionnalité intéressante d'AssertJ est *assertThatThrownBy* qui permet de vérifier qu'une exception est bien levée (et qui permet même de vérifier la classe, la cause, le message, etc. . .).
- Lors de la rédaction de mes tests, j'ai respecté l'organisation suivante : Les découper en 3 étapes principales :
  - GIVEN : Étape d'initialisation des données qui seront utiles pour le test. Pour garder un maximum de clarté, cette étape ne doit pas faire plus de 3 lignes. Ainsi, s'il le faut, on crée une nouvelle fonction pour initialiser les variables.
  - THEN : Ici on exécute les fonctions que le test a réellement pour but de tester. Si par exemple on désire tester la fonction qui génère les logs, c'est ici que l'on va l'appeler et

récupérer le log qu'elle a produit.
— WHEN : Enfin, c'est là que l'on réalise toutes les assertions pour vérifier que la fonction appelée précedemment a produit le résultat attendu.

La rédaction des tests unitaires doit toujours respecter la même règle : Obtenir un taux de couverture de lignes supérieur à un certain pourcentage. La couverture des lignes désigne la part des lignes du code ayant été lues lors de l'exécution de tous les tests unitaires, et chez Worldline ce taux minimum est fixé à 80%. Les fonctions que j'ai écrites n'étant pas très longues (grâce au fait que les fonctions principales sont récursives), j'ai pu obtenir 95% de couverture relativement facilement.

4.4.2 Tests sur les logs

Afin de tester les nouvelles méthodes que j'avais implémentées, j'ai été amené à réaliser des tests sur les logs générés par le module. Pour tester le résultat des logs produits par mes méthodes, il fallait pouvoir y accéder au sein des tests, soit en dehors des fichiers dans lesquels on écrit d'habitude. C'est donc ici le rôle de ce qu'on appelle un « log appender ». Un appender est un objet qui est principalement responsable de rediriger des messages vers différentes destinations telles que la console, des fichiers, des sockets, ... Il m'a ainsi fallut modifier la configuration des logs présente dans logback.xml pour ajouter une nouvelle configuration d'appender pour que je puisse l'utiliser dans mes classes de test.
Une fois cela fait, il ne me restait plus qu'à déclarer un nouvel objet appender dans mes tests et m'assurer que les mes méthodes produisaient les résultats attendus.

Chapitre 5

Second sujet : Docker

5.1 Rappels

Voici quelques rappels sur les principaux concepts de docker que j'ai été amené à utiliser :

— **Docker :** Contrairement aux hyperviseurs de machines virtuelles, les isolateurs (dont fait partie Docker) ne reposent pas sur une émulation du matériel physique. Ils sont de ce fait très performants lorsqu'il s'agit d'isoler l'exécution d'applications. Docker est donc un isolateur permettant de lancer des applications au sein de conteneurs logiciels.

— **Conteneur :** Le rôle d'un conteneur est d'héberger des services sur un même support physique tout en les isolant les uns des autres. Un conteneur ne requiert pas de système d'exploitation et utilise uniquement les fonctionnalités du noyau, rendant ainsi les images docker indépendantes du système d'exploitation d'où elles sont créés/hébergées.

— **Image :** Une image Docker est un fichier composé de plusieurs couches qui est utilisé pour exécuter des applications dans un conteneur. Une image est construite à partir d'instructions définies dans un fichier nommé Dockerfile. Il est possible de stocker ces images dans un dépôt (ou repository) publique ou privé afin que d'autres utilisateurs puissent déployer des conteneurs, tester ou partager ces images.

— **Dockerfile :** Les Dockerfiles sont donc les fichiers permettant de construire les images Docker. On va essentiellement y définir l'image de base de notre conteneur, exécuter des commandes pour installer des logiciels spécifiques aux besoins de l'image et configurer par exemple le répertoire courant, ajouter des fichiers de configuration ou encore exposer des ports.

— **Docker compose :** Docker compose est un outil destiné à créer et lancer des applications sur plusieurs conteneurs à la fois. Il utilise des fichiers YML pour configurer les services de l'application et performe la création et le lancement de tous les conteneurs en une seule commande.

— **Docker Swarm :** Permet de grouper un ensemble de machines physiques ou virtuelles sur lesquelles l'application Docker est active. Lorsque ces machines sont configurées pour se rejoindre en cluster, il est possible d'envoyer des commandes Docker à toutes les machines du cluster. Les machines sont alors contrôlées par un manager et on désigne les machines par nœuds ou nodes.

— **Portainer :** Portainer est essentiellement un petit conteneur permettant de superviser d'autres conteneurs. Portainer dispose d'une interface web pour gérer simplement ces conteneurs. On peut entre-autres : Superviser, créer, supprimer, éditer, redémarrer, ...

5.2 Le module tms-wp

Le module sur lequel j'ai été amené à travailler se prénomme donc TMS-WP, qui est l'abréviation de Transaction Management Service - Welcome Point. Le rôle de ce module est de recevoir les transactions émises par les deux FO (ou Front Office qui sont les interfaces que les marchands utilisent pour réaliser toutes les opérations dont ils ont besoin). Ils peuvent par exemple demander à annuler, confirmer ou encore rechercher une transaction.

Voici un schéma simplifié représentant l'architecture logique actuelle l'application :

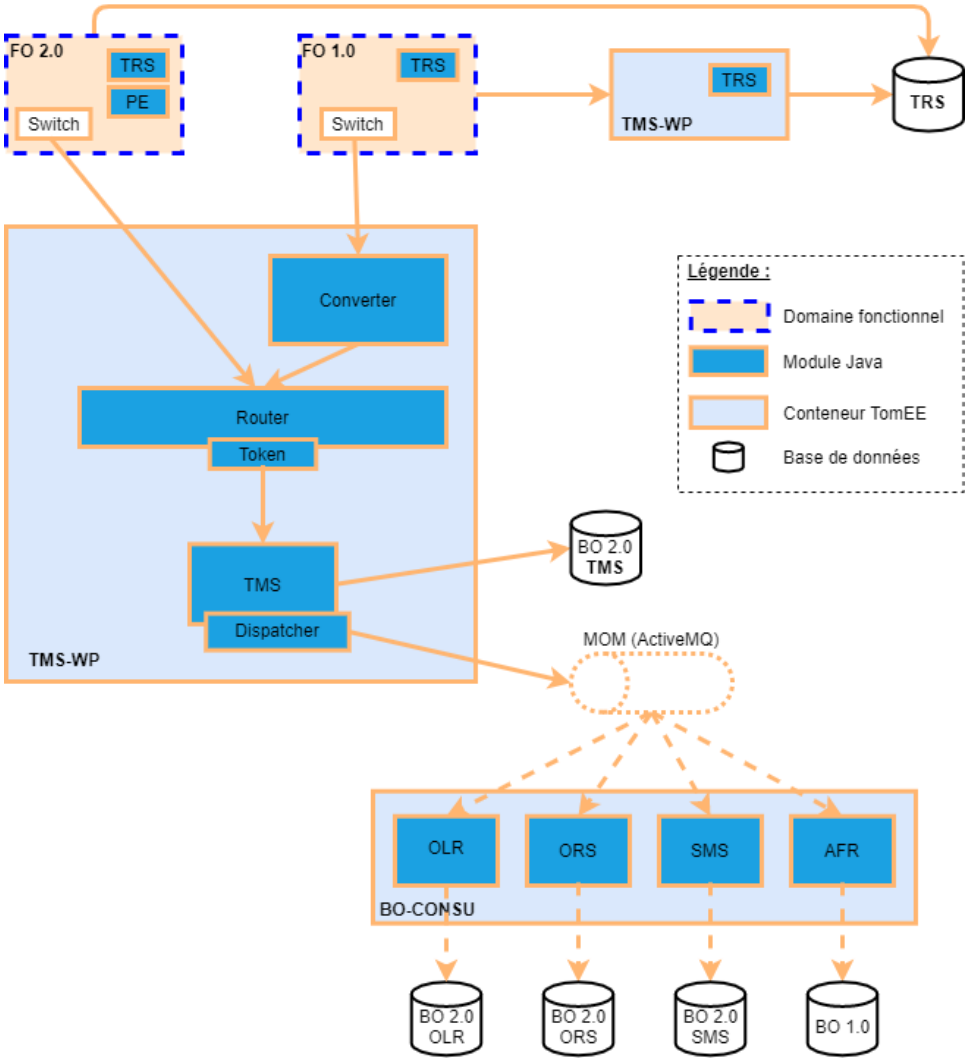

FIGURE 5.1 – Schéma simplifié de l'application TMS

Le module FO2 appelle ainsi la partie « Router » de TMS-WP, tandis que le module FO1 va d'abord passer le converter car certains champs qu'il émet ont besoin d'être traduits afin d'être reconnus par les autres modules.
Les données sensibles (numéros de cartes, adresses email, ...) sont alors chiffrées à l'aide du module token avant d'être transmises au module TMS qui va s'occuper de deux choses :

— Traiter la transaction à partir des données de la base TMS (sauvegarder, modifier, ...)
— Envoyer ces données au module dispatcher qui va les transmettre à de plus petites bases dont le but est de permettre d'accéder plus rapidement à ces données en les classant plus précisément et dans des bases de taille réduite comparé à la base principale TMS.

Une fois cela fait, le module TMS renvoie une réponse vers le marchand.

Comme toutes les applications développées à Worldline, celle-ci intègre Jenkins. C'est un outil open-source construit pour le développement continu. En quelques mots, voici les principales fonctionnalités de Jenkins : Après chaque commit, il récupère le nouveau code, construit et teste le projet. Ainsi les développeurs n'ont plus qu'à se concentrer sur le dernier commit qui a fait échouer les tests, ce qui permet de créer plus souvent des nouvelles versions de l'application. Un autre avantage est l'automatisation des tests : En effet, l'outil Jenkins automatise toute l'exécution des tests et le déploiement de l'application et le développeur n'a plus qu'à commit ses changements pour que son code soit testé et déployé. Cela même si c'est un conteneur Docker par exemple.

Ainsi, périodiquement et après chaque commit sur le projet Gitlab, Jenkins va suivre les étapes suivantes en affichant le temps passé sur chacune :

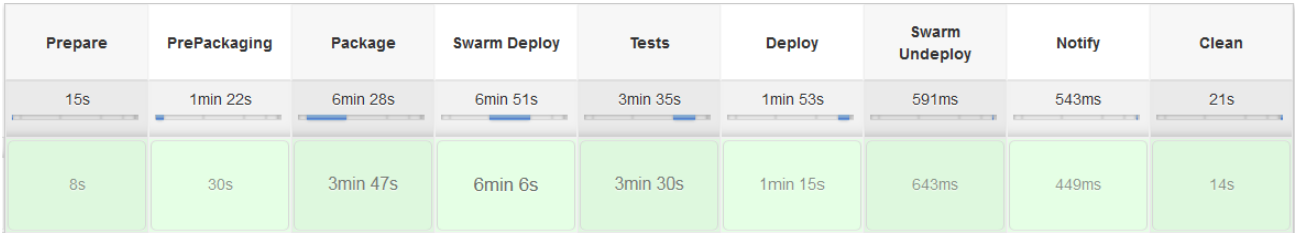

FIGURE 5.2 – Exemple de résultat de build Jenkins

Cette application est organisée de la même manière que la plupart des applications de Worldline. Elle est découpée en plusieurs modules qui ont chacun un rôle précis. Voici les principaux modules qui la composent :

— **Core :** Le core correspond à la partie DAO de l'application. Pour résumer, c'est un patron de conception qui permet de s'abstraire de la façon dont les données sont stockées pour y accéder. On va donc y retrouver les méthodes de gestion des données (accès, persistance, ...) qui seront utilisables sans avoir besoin de connaître les spécificités de la base de données.
— **Contrat :** Le module contrat possède deux fonctions principales. Mapper les objets du core et déclarer le service de l'application. En effet comme les projets sont en général des webservices, on déclare un service qu'il faudra implémenter et dont les méthodes seront appelées depuis l'extérieur.
— **Provider :** Ce module correspond au contrôleur de l'application. Il valide les objets selon certaines contraintes.
— **Mock :** La partie mock va reproduire le comportement de l'application, notamment en implémentant la partie contrat afin qu'il soit possible de s'en servir indépendamment des services qu'elle utilise normalement.
— **Test :** Contient les tests unitaires de l'application. Ces tests seront automatiquement lancés par Jenkins, et leur résultat sera utilisé pour alimenter diverses statistiques.

5.3 La problématique

Comme nous l'avons vu dans la section 5.2, l'application TMS-WP est centrale puisque toutes les transactions y passent à un moment donné. Très peu d'opérations sont effectuées sur les données puisque le rôle de cette application est de transmettre les données aux modules qui en ont besoin. Cependant, son rôle est important et elle mérite d'être testée intégralement. Il existe bien sûr des tests unitaires qui couvrent actuellement 64% du code, mais aucun test d'intégration n'a encore été mis en place.

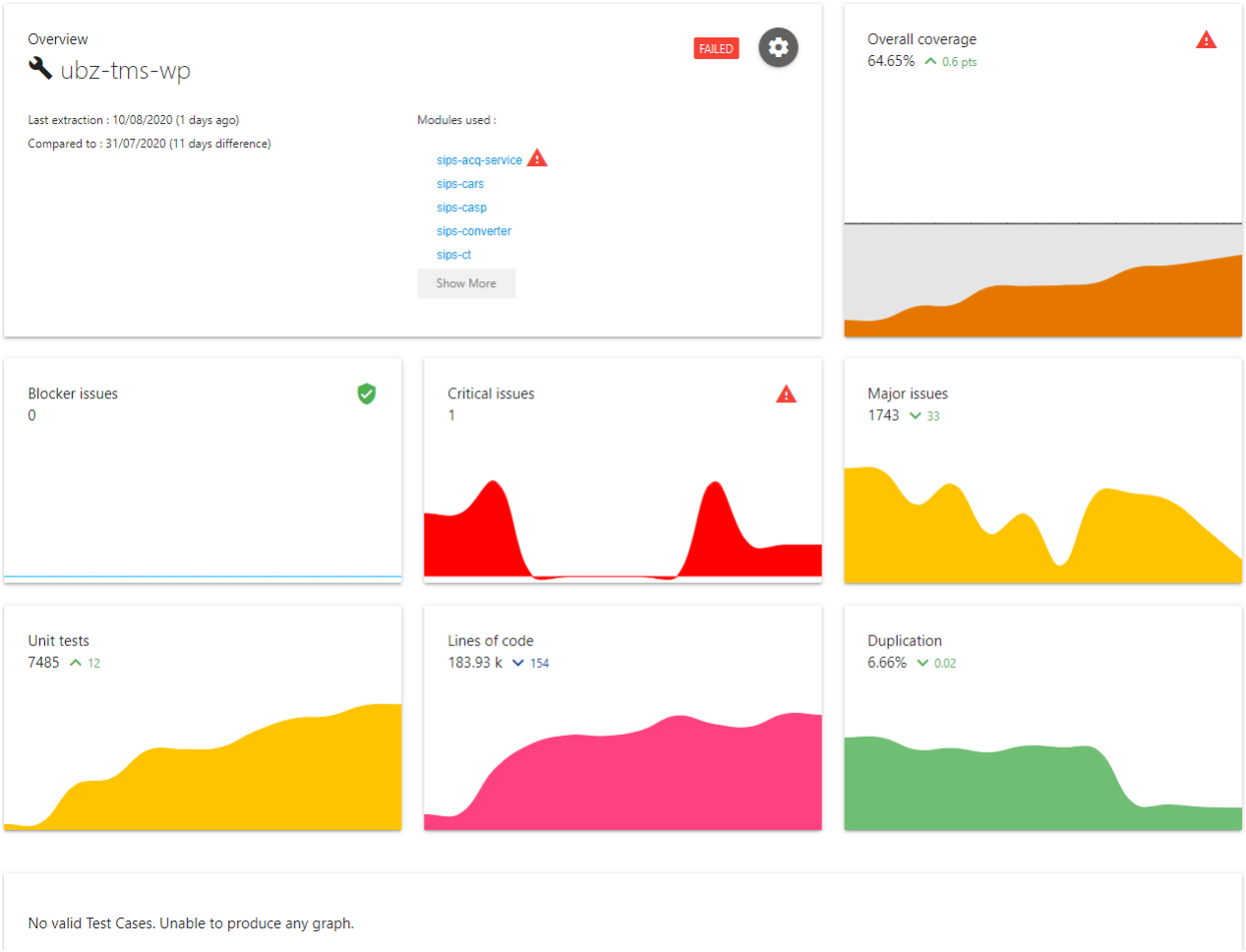

FIGURE 5.3 – Le suivi du projet tms-wp par l'application submarine de Worldline

L'application submarine produit différent graphiques permettant de facilement suivre l'évolution de l'état des applications de Worldline. On peut voir sur l'image ci-dessus que l'application tms-wp est actuellement stable et possède peu d'erreurs critiques, mais aucun test d'intégration n'est exécuté.

Ce type de test possède l'intérêt de permettre aux développeurs de détecter bien plus tôt au sein du cycle de déploiement de l'application des erreurs qui n'ont pas été révélées par les tests unitaires. Ce qui leur fera gagner du temps puisque l'impact de ces erreurs sera réduit et puisqu'en général les développeurs sont passés sur un nouveau sujet au moment où les erreurs apparaissent

lors du déploiement.

Cependant il n'est actuellement pas possible de réaliser ces tests d'intégration tant que l'application n'est pas Dockerisée puisqu'il est nécessaire de pouvoir réaliser des appels au webservice de tms-wp qui doit lui-même être hébergé sur un serveur JBoss. La base de données de tms-wp est une base oracle qui a également besoin de fonctionner dans un conteneur.

Le but serait donc de mettre en place des tests d'intégration qui soient exécutés automatiquement afin de pouvoir suivre l'évolution de leur résultat et de leur couverture. Voici un exemple de graphique généré par l'application « Submarine » de Worldline qui suit tous les projets disponibles sur le GitLab interne à partir des Cucumber reports :

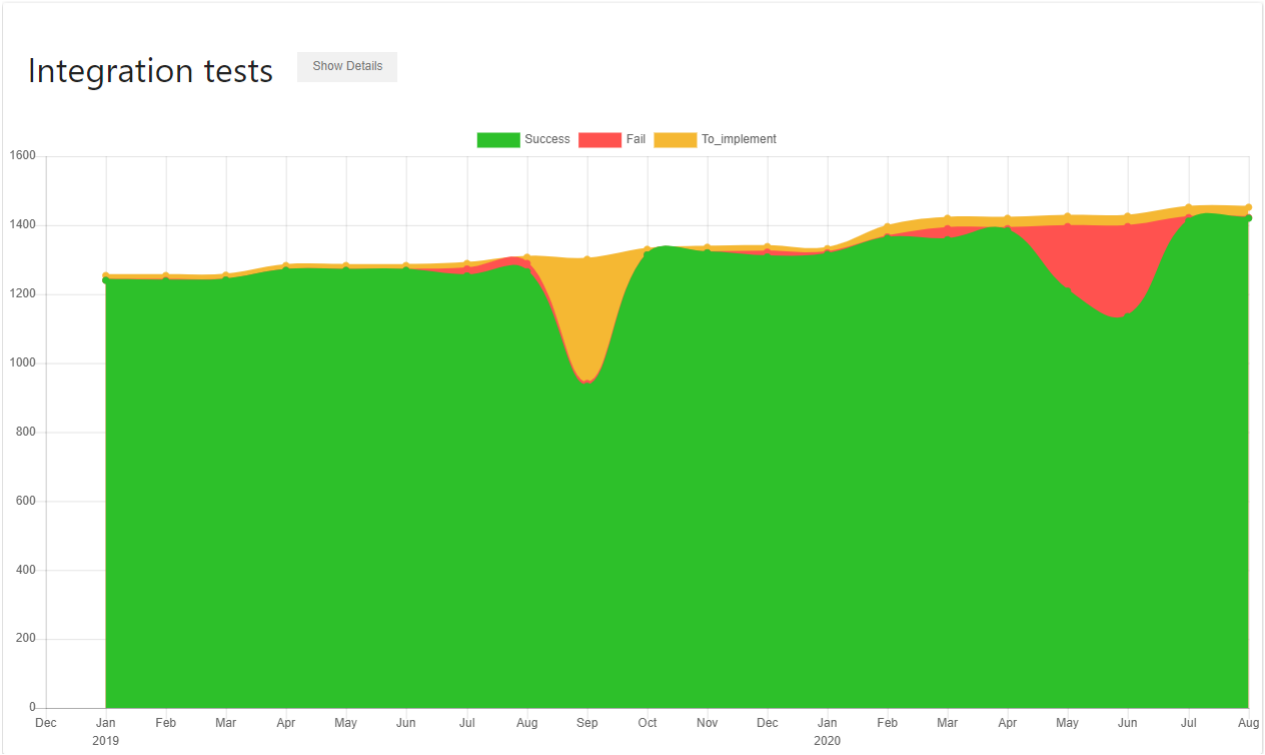

FIGURE 5.4 – Illustration : Tests d'intégration du module M2M au cours du temps

À la fin de ce projet, le même type de graphique devra être généré automatiquement et régulièrement à partir des tests d'intégration rédigés.

5.4 Dockerisation de l'application tms-wp

La solution pour pouvoir exécuter ces tests d'intégration sans avoir à complétement déployer l'application tms-wp est donc de la dockeriser. De cette manière, après chaque changement sur tms-wp, il sera possible de la déployer localement au sein d'un conteneur et de lancer une série de tests qui appelleront les méthodes du webservice tms-wp pour vérifier que son comportement est correct.

L'application TMS est structurée en trois projets principaux :

— **sips-tms :** C'est le cœur de l'application tms. Comme je l'ai expliqué plus haut, c'est ce projet qui contient les packages core, contrat, provider, ... On y trouve donc toute la logique métier.

— **sips-delivery-tms-wp :** Son rôle est de construire le livrable de l'application tms. Il est donc majoritairement constitué de fichiers de configuration, c'est pourquoi on y retrouvait initialement les deux modules app et conf (c'est donc au sein de cette application que j'ai passé le plus de temps).

— **sips-delivery-bdd-tms :** Ce projet sert à construire le livrable de bdd-tms. On y retrouve tous les scripts de déploiement de la base de données ainsi que le nécessaire pour construire, déployer et tester l'image Docker de la base.

Pour pouvoir déployer tms-wp dans un conteneur, il faut également déployer les principales applications sur lesquelles elle repose afin qu'elle fonctionne correctement. Ainsi, lorsqu'on lance le déploiement docker, on ne crée pas un simple conteneur mais une stack de docker swarm (voir les rappels section 5.1) dans laquelle on va trouver les conteneurs suivants :

— tms-wp-hazelcast : Échange de messages client-serveur.

— tms-wp-sips-settlements : Une base de données servant à une ancienne version de sips. C'est pourquoi elle a depuis été mockée pour accélérer le déploiement.

— tms-wp-sips-trs : Même idée que settlement (mockée également).

— tms-wp-sn-sips-tms : Base de données de tms dans laquelle sont stockées les transactions. C'est sur cette base que nous vérifierons si les données ont bien été sauvegardées par TMS.

— tms-wp-ubz-tms-wp : L'application tms elle-même.

```
docker@boot2docker:~$ docker service ls
ID                  NAME                MODE                REPLICAS
ysprzucbxeg0        tms-wp_hazelcast    replicated          1/1
mhab0c1zvkee        tms-wp_sn-sips-tms  replicated          1/1
gj5p9mymd7o7        tms-wp_ubz-tms-wp   replicated          1/1
```

FIGURE 5.5 – Les services déployés après avoir lancé la stack de tms-wp

Le processus de dockerisation s'est déroulé de la manière suivante : Il a d'abord été créé au sein du projet sips-delivery-tms-wp un nouveau module intitulé sips-delivery-tms-wp-docker afin de pouvoir paramétrer proprement la configuration Docker. Ce nouveau module contient trois sous modules :

— **docker-app :** Récupère la configuration du module tms-wp-app.

— **docker-conf :** Récupère la configuration du module tms-wp-conf.

— **docker-image :** Possède tous les fichiers nécessaires au déploiement de tms-wp sous docker (donc on y trouve par exemple les fichiers Dockerfile, docker-compose.yml, ...).

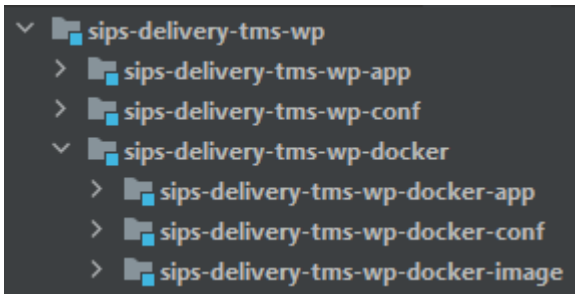

FIGURE 5.6 – Les différents modules de sips-delivery-tms-wp

Ces trois modules vont chacun être compilés et avoir un livrable de construit, c'est pourquoi on retrouve dans tous ces modules le fichier docker-assembly.xml qui va indiquer à Maven quels sont les fichiers qu'il faut inclure dans le livrable. Ainsi comme le module docker-image représente le « produit final » de la construction du livrable de l'application docker, on retrouve dans son fichier assembly des inclusions de fichiers des deux autres modules app et conf. On va donc d'abord compiler les modules app et conf avant de se servir de leurs livrables qui viennent d'être produits pour construire celui de docker image.

Si on veut redéfinir des fichiers provenant d'un module et que l'on empaquette dans le livrable d'un autre module, il faut d'une part les exclure au sein du fichier assembly et d'autre part créer notre nouvelle version dans le module dont est en train de créer le livrable.

```
<fileSets>
    <fileSet>
        <directory>${basedir}/../../sips-delivery-tms-wp-conf/src/main/config/template/properties</directory>
        <outputDirectory/>
        <includes>
            <include>*/**</include>
        </includes>
        <excludes>
            <exclude>**/resource-bindings.properties</exclude>
            <exclude>**/logback.xml</exclude>
        </excludes>
    </fileSet>

    <!-- Docker specific resource-bindings.properties -->
    <fileSet>
        <directory>${basedir}/src/main/config/template/properties/</directory>
        <outputDirectory/>
    </fileSet>
</fileSets>
```

FIGURE 5.7 – Exemple d'inclusion/exclusion dans un fichier assembly

Ci-dessus est un exemple du système d'exclusion/inclusion dans un fichier assembly : On copie d'abord dans le premier <fileset> les fichiers du dossier properties du module tms-wp-conf dans le livrable du module tms-wp-docker-conf tout en excluant le fichier resource-binding.properties

que l'on désire redéfinir. Puis dans le second <fileset>, on copie le contenu du dossier properties du module tms-wp-docker-conf (qui possède donc une nouvelle définition du fichier resource-binding.properties) dans le livrable en construction.
Pour illustrer l'utilité d'un tel système, on peut continuer avec l'exemple de resource-binding.properties : Ce fichier permet d'indiquer quelle implémentation de telle fonction on désire utiliser. Cela permet par exemple d'indiquer que l'on ne veut pas utiliser l'implémentation du webservice de sips-settlements définie dans le module provider, mais celle que l'on a créée dans notre nouveau module mock spécifiquement pour notre application docker.

Comme vous l'avez compris, il y a donc un ordre à respecter lors de la génération des livrables, puisqu'un livrable peut avoir besoin des données d'un autre livrable au sein du même projet. Cet ordre est automatiquement détecté par reactor qui est une fonctionnalité de Maven qui va déterminer l'ordre de construction à partir des diverses dépendances déclarées dans tous les modules du projet.

```
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Build Order:
[INFO]
[INFO] Sips tms wp - Delivery                                             [pom]
[INFO] Sips tms wp - Delivery - App                                       [pom]
[INFO] Sips tms wp - Delivery - Conf                                      [pom]
[INFO] Sips tms wp - Delivery - Docker                                    [pom]
[INFO] Sips tms wp - Delivery - Docker - App                              [pom]
[INFO] Sips tms wp - Delivery - Docker - Conf                             [pom]
[INFO] Sips tms wp - Delivery - Docker - Image                            [pom]
[INFO] Sips tms wp - Delivery - Tests                                     [jar]
```

FIGURE 5.8 – L'ordre de construction du projet sips-delivery-tms-wp déterminé par reactor

Voici quelques fichiers essentiels à la dockerisation de l'application tms-wp :

— **Docker-compose.yml :** Ce fichier permet de définir et de lancer l'application Docker multi-conteneurs (donc la stack de docker swarm dans notre cas précis).

```yaml
version: '3'

services:
    ubz-tms-wp:
        image: registry-testing.kazan.atosworldline.com/sandbox/awl-ubz-tms-wp:${USERNAME}
        depends_on:
            - sips-trs
            - sn-sips-tms
        ports:
            - "8080:8080"

    sn-sips-tms:
        image: registry-testing.kazan.atosworldline.com/sandbox/awl-sn-sips-tms:latest
        ports:
            - "1521:1521"

    hazelcast:
        image: registry-testing.kazan.atosworldline.com/sandbox/awl-ubz-hazelcast:3.9.3
```

FIGURE 5.9 – Le fichier docker-compose.local.yml

Une fois ce fichier défini, il est possible de déployer l'application multi-conteneurs en une seule commande : docker stack deploy. Il existe une version locale (voir ci-dessus) et une version jenkins où l'on va définir en plus sur quel réseau les conteneurs déployés devront se trouver.

— **deploy-to-local-swarm.bat :** Ce script permet de lancer en une ligne toutes les étapes du déploiement de l'application tms-wp sous docker, ce qui a été spécialement utile lors du développement. Les principales étapes qu'il suit sont les suivantes : Construire le projet avec les toutes dernières modifications, construire l'image docker de tms-wp à partir du projet qui vient d'être créé et enfin lancer la stack docker avec tous les conteneurs nécessaires (dont le conteneur de l'image de tms-wp que l'on vient de construire bien sûr).

— **Jenkinsfile :** Ce fichier définit toutes les propriétés dont Jenkins a besoin pour correctement exécuter toutes ses étapes.

Mon rôle a initialement été de tracer l'origine des erreurs lors du déploiement du premier jet d'application dockerisée, ce qui était pour moi une bonne manière de me familiariser avec le fonctionnement de l'application tms-wp et m'a fait de bons rappels sur le fonctionnement de Docker.

Comme l'implémentation de l'interface du webservice de l'application sips-settlement dont l'application sips-tms se sert a besoin d'accéder à la base de données sips-settle, nous avons dans un premier temps ajouté le conteneur de la base sips-settle en plus à la stack docker afin que l'on puisse initialiser notre application sans erreur.
La deuxième étape a donc été de remplacer par une version mockée l'implémentation de l'interface de sips-settlement, ce qui nous évitait de nous connecter à la base sips-settle, nous permettant ainsi de supprimer l'image de cette base de données de la stack docker. En effet, on ne souhaite pas tester le comportement de sips-tms avec sips-settlement comme je l'ai dit dans la section 5.4. Comme l'application sips-settlement ne disposait pas de mock de son interface, je me suis chargé de le créer. J'ai donc ajouté un nouveau module sips-settle-mock qui possède une unique classe

qui implémente l'interface en question : SettlementManagementServiceImpl.

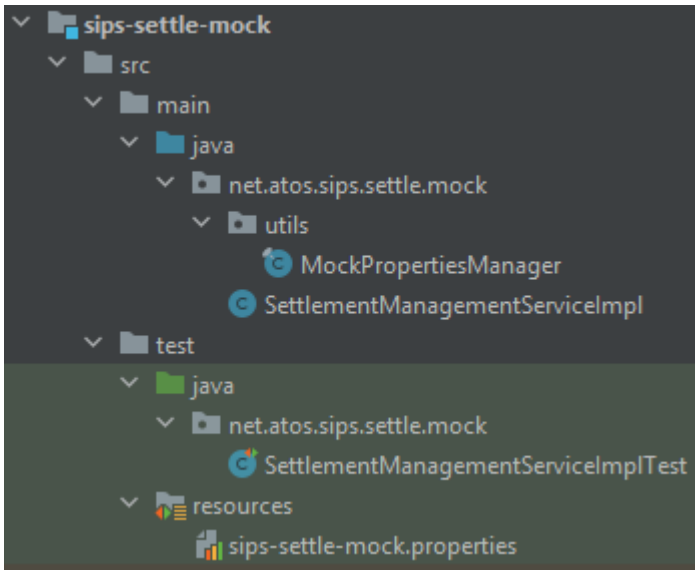

FIGURE 5.10 – L'architecture du module sips-settle-mock

Comme on peut le voir dans le dossier test, j'ai également rédigé des tests unitaires à partir de simples cas d'utilisation que l'on m'a fournis (à partir d'une certaine valeur en entrée, il faut renvoyer un certain objet).

5.5 Réalisation de tests d'intégration

La mise en place des tests d'intégration représente la plus grosse partie de mon stage puisqu'elle m'a occupée presque 4 semaines sur les 10 semaines de mon stage. Je suis cette fois ci parti d'un projet vide et j'ai moi-même entièrement codé ce nouveau module.

Voici l'arborescence du projet à la fin de mon stage :

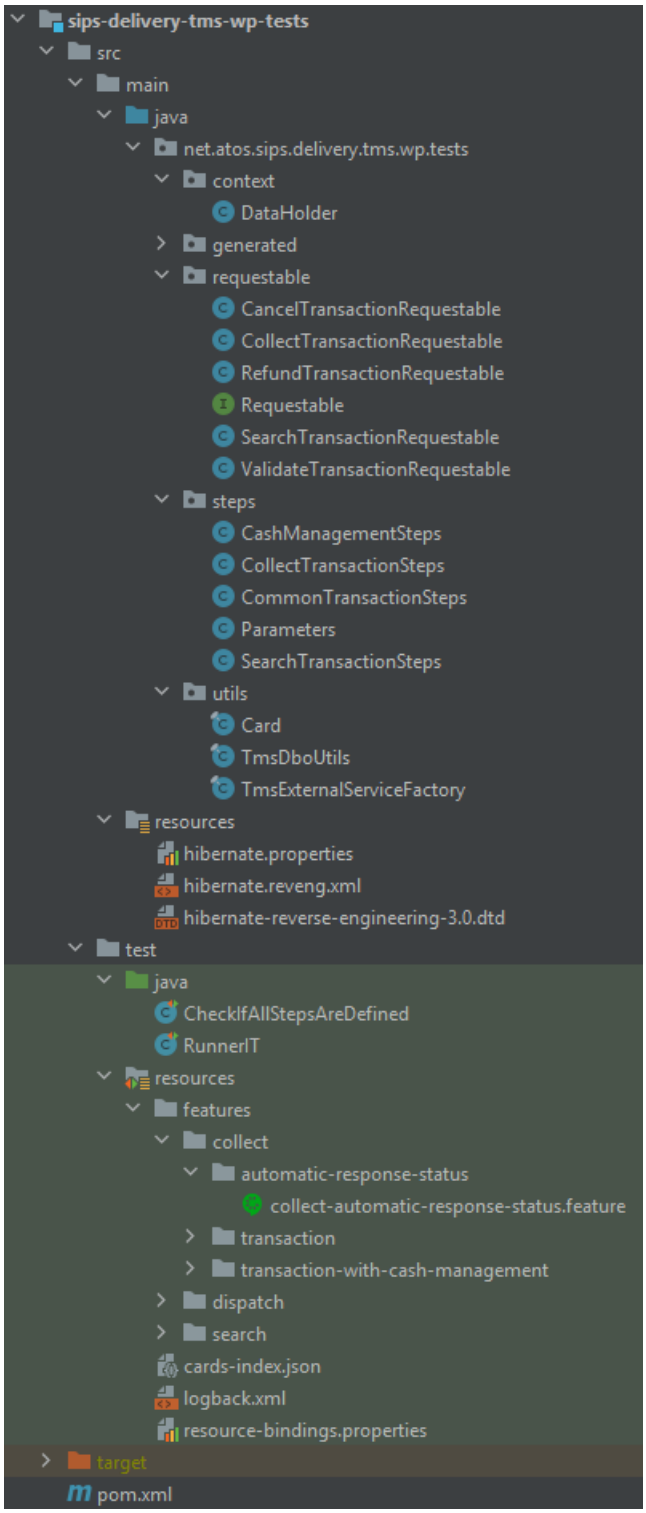

FIGURE 5.11 – L'architecture du module sips-delivery-tms-wp-tests

Il est structuré en 2 paquets principaux : *main* qui contient tout le code java et *test* qui contient principalement les fichiers .feature de cucumber. En effet les tests sont lancés par l'outil Cucumber qui permet de spécifier le comportement et les étapes des tests dans un langage facile à comprendre constitué de phrases commençant par given, then, when, ...

Avant de rentrer dans les détails de la structure de l'application de test, voici les définitions des concepts principaux de Cucumber :

- **Les features :** Une feature est un scénario d'utilisation qui décrit le comportement des tests. On crée donc des fichiers .feature dans lesquels on peut définir un ou plusieurs scénarios. On définit également un ou plusieurs tags pour pouvoir les repérer par la suite.
- **Les step definitions :** Une step definition correspond au code attaché aux étapes définies dans les features. Elles définissent donc comment les étapes doivent être réalisées, c'est là que l'on trouve le code java rattaché aux tests.
- **Les runners :** Les runners font le lien entre les features et les step definitions. Ils commencent par parser les features, puis ils exécutent dans l'ordre les steps definitions correspondantes. Dans le tag @CucumberOptions que l'on peut associer aux runners, on peut définir les éléments suivants : glue permet à Cucumber de localiser le fichier des step definition, tags permet de spécifier quels scénarios exécuter à partir de ce runner en fonction des tags des scénarios du fichier .feature, strict est un booléen qui, si vrai, indique que l'exécution des tests doit échouer si on rencontre un step sans step definition. Enfin dryRun est un booléen qui va indiquer à Cucumber qu'il ne doit que vérifier que toutes les steps possèdent une step definition (il n'exécute donc pas les tests à partir d'un runner avec dryRun=true).
- **Picocontainer :** Le concept de picocontainer est relativement complexe, mais la raison pour laquelle j'ai été amené à m'en servir est qu'il permet entre autre de passer en paramètre des objets aux constructeurs des classes des steps definitions, ce qui est impossible nativement avec Cucumber.

Avant de modéliser la structure du projet, j'ai participé à une réunion où les delivery manager du projet ont défini les différents scénarios qu'il faudrait implémenter à minima, à la fin de laquelle le document suivant a été produit :

Scénario (1 ligne = 1 scénario)					
	settlement option				Priorité
Collect	AUTHOR_CAPTURE	standard			1
		hotlist			
	IMMEDIATE				2
Collect + Cash Management	AUTHOR_CAPTURE	standard		1 cancel partiel	3
				1 cancel total	4
				2 cancels partiels (=> avec un reste à remiser)	
				1 cancel partiel + 1 cancel partiel (=> cancel total)	
		hotlist		1 cancel partiel	
				1 cancel total	
				2 cancels partiels (=> avec un reste à remiser)	
				1 cancel partiel + 1 cancel partiel (=> cancel total)	
	IMMEDIATE			1 cancel partiel	
				1 cancel total	
				2 cancels partiels (=> avec un reste à remiser)	
				1 cancel partiel + 1 cancel partiel (=> cancel total)	
				1 Validation partielle	6
				1 validation totale	5
				1 Validation partielle suivie d'1 cancel	
				1 cancel partiel suivi d'1 validation	
	Refund		transaction en settle_ok	1 rmbt partiel	7
				1 rmbt partiel puis 1 rmbt partiel (avec un montant =0)	
				1 rmbt total	8
				1 rmbt partiel puis 1 rmbt partiel (avec un montant final =0)	
Search			nécessite au préalable de créer une transac		10
Réponse automatique		transaction existe			30
		transaction n'existe pas	resp auto dans "orphan"		31
			transaction créée après resp auto	resp auto supprimé de "orphan", et transaction maj	32
Dispatch			Pour tous les types d'opérations/transactions, les 4 files amq doivent ou non être alimentées + assertion sur champ MessageObject Note : OLR est toujours alimenté contrairement aux	cf tous les use case précédents	20

FIGURE 5.12 – Le tableau des cas d'utilisation

On note ici que le but de mon projet était surtout de mettre en place une base permettant de facilement et clairement rédiger des tests et non de mettre en place un maximum de tests.

Pour éviter de redéfinir plusieurs fois le même test puisque les tests seront tous plus ou moins similaires, j'ai adopté la structure suivante :

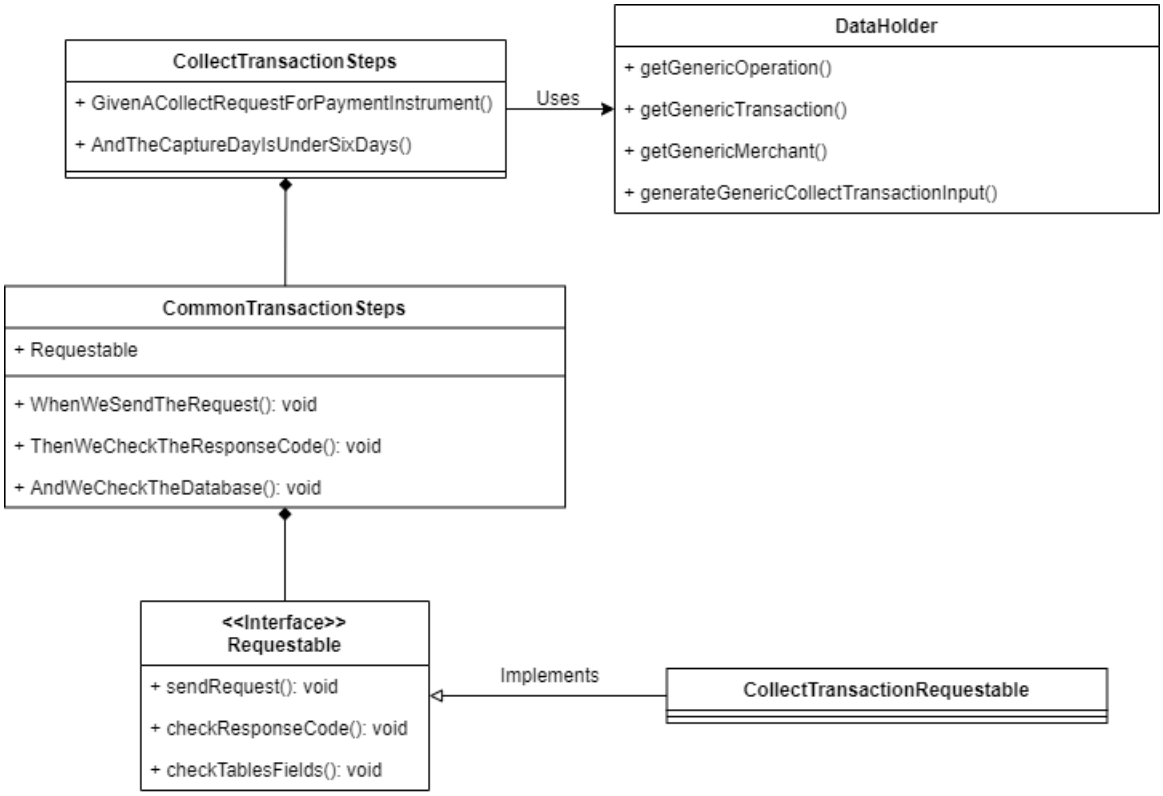

FIGURE 5.13 – Structure des tests d'intégration de tms-wp

Voici le fonctionnement complet des tests ainsi rédigés : Une fois lancé à partir du **runner**, Cucumber va lire tous les fichiers **feature** et les examiner pour trouver les **scénarios** à exécuter. Enfin Cucumber va relier ces scénarios aux **steps definitions** qui sont le code Java qui sera exécuté. À ce moment, les classes de test comportant les steps definitions seront instanciées à l'aide de **Picocontainer**. Les tests seront alors exécutés par **JUnit** qui est un framework permettant d'automatiser les tests (on se sert également d'**AssertJ** pour rédiger les assertions comme dans le premier sujet de mon stage afin de rendre les tests plus lisibles).

Pour éviter d'avoir à écrire des fonctions très similaires lorsque les phrases des scénarios se ressemblent, on utilise la fonctionnalité des **paramètres** Cucumber. Ces derniers offrent la possibilité de définir des variables au sein des phrases, ce qui donne ce genre de scénarios :

```
Feature: Call CollectTransaction in AUTHOR_CAPTURE mode then call some cash management methods

  Scenario: Call CollectTransaction in AUTHOR_CAPTURE mode then partially cancel it
    Given a successful 10 EUR collect request for CB payment instrument in AUTHOR_CAPTURE mode
    And a 7 EUR cancel request
    When we send the request
    Then we check the response code is OK
    And we check that the transaction was inserted in the database

  Scenario: Call CollectTransaction in AUTHOR_CAPTURE mode then totally cancel it
    Given a successful 10 EUR collect request for CB payment instrument in AUTHOR_CAPTURE mode
    And a 10 EUR cancel request
    When we send the request
    Then we check the response code is OK
    And we check that the transaction was inserted in the database
```

FIGURE 5.14 – Exemple de scénarios avec des paramètres (en bleu)

Les vérifications du contenu de la base de données de tms-wp se font via le framework **Hibernate**. Celui-ci permet de récupérer les objets en base de données sans avoir besoin d'écrire des requêtes SQL. En revanche Hibernate requiert de créer des classes correspondant exactement à la structure des tables de la base de données. Les tables de la bdd de tms-wp étant nombreuses et possèdant beaucoup de colonnes, il aurait été très long de les créer à la main. Ainsi, j'ai utilisé le plugin **hibernate tools** pour générer automatiquement le code Java des classes correspondant aux différentes tables de la base de données. Comme hibernate-tools est un plugin, il sera possible de le relancer pour qu'il modifie directement le code source du projet de façon à ce que les classes correspondent à la base de données dans le cas où elle aurait été modifiée.
On peut noter ici que comme toutes les tables de la bdd de tms ne sont pas utiles, on utilise un fichier hibernate.reveng.xml pour ne sélectionner que les tables du schéma TMS_DBO qui nous intéresse et ainsi éviter de générer des centaines de classes inutiles.

Comme les tests doivent pouvoir être exécutés sans erreurs en local et par Jenkins, il faut pouvoir faire varier les adresses utilisées pour se connecter au webservice de tms-wp et à sa base de données. Pour cela, on utilise la fonction Java System.getenv(...) et on regarde si des variables d'environnement ont été injectées ou pas et si ce n'est pas le cas, l'adresse est localhost. En effet lorsque Jenkins lance les différents conteneurs, il définit des variables d'environnement sur les machines qui lanceront les tests afin de pouvoir accéder aux autres conteneurs Docker.

Concernant Jenkins, il va exécuter les étapes de la figure 5.2 périodiquement (tous les jours entre minuit et 7h) et pousser une version *latest* du projet tms-wp sur le registry de Worldline si toutes les étapes ont pu s'exécuter sans erreur.

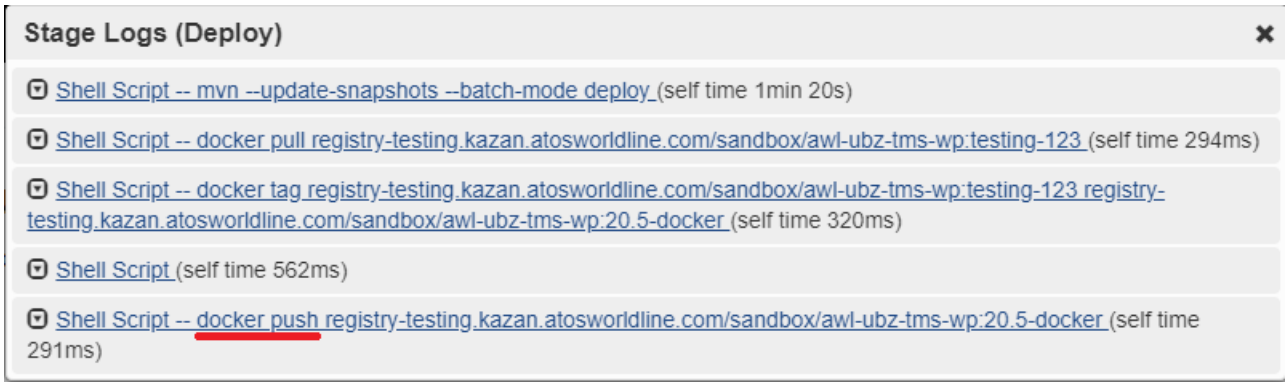

FIGURE 5.15 – L'étape « deploy » de Jenkins

Ci-dessus la dernière étape de Jenkins « deploy » qui pousse la dernière version de tms-wp sur le registry de Worldline.

5.6 Exemples de problèmes rencontrés

— Lorsque j'appelais les méthodes du webservice tms-wp dockerisé, des erreurs liées à la base de données sn-sips-tms remontaient. En effet la base dont se sert tms-wp qui avait été dockerisée n'était pas identique à celle qui tournait en production : Certains champs étaient manquants ce qui fait que les appels à la bdd ne pouvaient pas aboutir. J'ai donc modifié les scripts SQL de création des tables afin qu'ils soient conformes à ce qui était présent en production. Ces erreurs n'étaient pas détectées car la base de données dockerisée n'avait pas été paramétrée pour être buildée et testée par Jenkins (ce qui a été corrigé une fois découvert). Une fois ces modifications effectuées, la base de données dockerisée était utilisable, et les appels au service tms-wp ne renvoyaient plus d'erreurs, j'ai donc pu reprendre la rédaction de mes tests d'intégration.

— La base de données oracle de tms affichait tout de suite le statut 'healthy' après avoir été créée alors qu'il n'était pas encore possible de s'y connecter, ce qui posait problème lors du déploiement sur Jenkins puisqu'il lançait immédiatement les tests et renvoyait une erreur. Il m'a donc fallu créer un script shell vérifiant que la connexion à la base était possible au sein du projet sips-delivery-bdd-sn-sips-tms qui était ensuite copié dans le conteneur de la base. Ce dernier renvoie le statut exit 0 si la connexion et possible et exit 1 sinon, il est donc possible d'ajouter la ligne suivant dans le dockerfile de la base :
HEALTHCHECK –start-period=30s –interval=20s –retries=5 CMD /usr/sbin/healthcheck.sh
On teste donc la connexion après 30sec toutes les 20 sec, 5 fois maximum (donc échec de lancement de l'image docker au bout de 2min10 si la connexion n'est toujours pas possible). Une fois que la connexion est possible, le conteneur passe du statut :

```
Up 2 minutes (healthy)
```

Au statut :

```
STATUS
Up 13 seconds (health: starting)
```

— Les POJOs (*Plain Old Java Object*) qui sont générées automatiquement par le plugin Hibernate-tools étaient initialement automatiquement générées à l'étape **generate-sources** du cycle de vie **compile** de Maven. On ne peut générer ces classes que lorsque l'accès à la base de données bdd-tms est possible, ainsi il faudrait que l'image de bdd-tms soit déjà déployée avant de compiler le projet tms-wp. Cela est malheureusement impossible avec Jenkins, ainsi il a été décidé de ne pas exécuter Hibernate-tools à l'étape generate-sources mais plutot de le définir comme plugin qu'il faudra lancer manuellement pour modifier le code source. Pour résumer : On doit commit une première fois les POJOs générés, puis, lorsque c'est nécessaire, on lancera le plugin Hibernate-tools qui modifiera le code source qu'il faudra ensuite commit pour prendre en compte les modifications.

Chapitre 6

Méthodologie et outillage

6.1 Technologies rencontrées

6.1.1 GitLab

GitLab est un dépôt Git basé sur le web qui fournit des repository publiques ou privés gratuits avec des fonctionnalités de suivi de problèmes et de creation de wikis. Il s'agit d'une plateforme DevOps complète qui permet aux professionnels d'exécuter toutes les tâches d'un projet, de la plannification du projet et de la gestion du code source à la surveillance et à la sécurité. Worldline dispose donc de son propre repository GitLab privé, accessible uniquement sur l'intranet où tous les projets de l'entreprise sont stockés.
J'ai donc utilisé GitLab tout au long de mon stage puisqu'à chaque modification d'un projet, je devais créer une nouvelle branche Git basée sur la version de la branche master sur laquelle j'effectuais mes modifications. Ensuite, pour que mes modifications soient réellement prises en compte dans le projet, je créais une *merge request* de ma branche sur master qui était ensuite passée en revue par la personne en charge du projet.
Voici un exemple de merge request sur le projet sips-settle lorsque j'ai mis en place un mock de l'interface de ce projet :

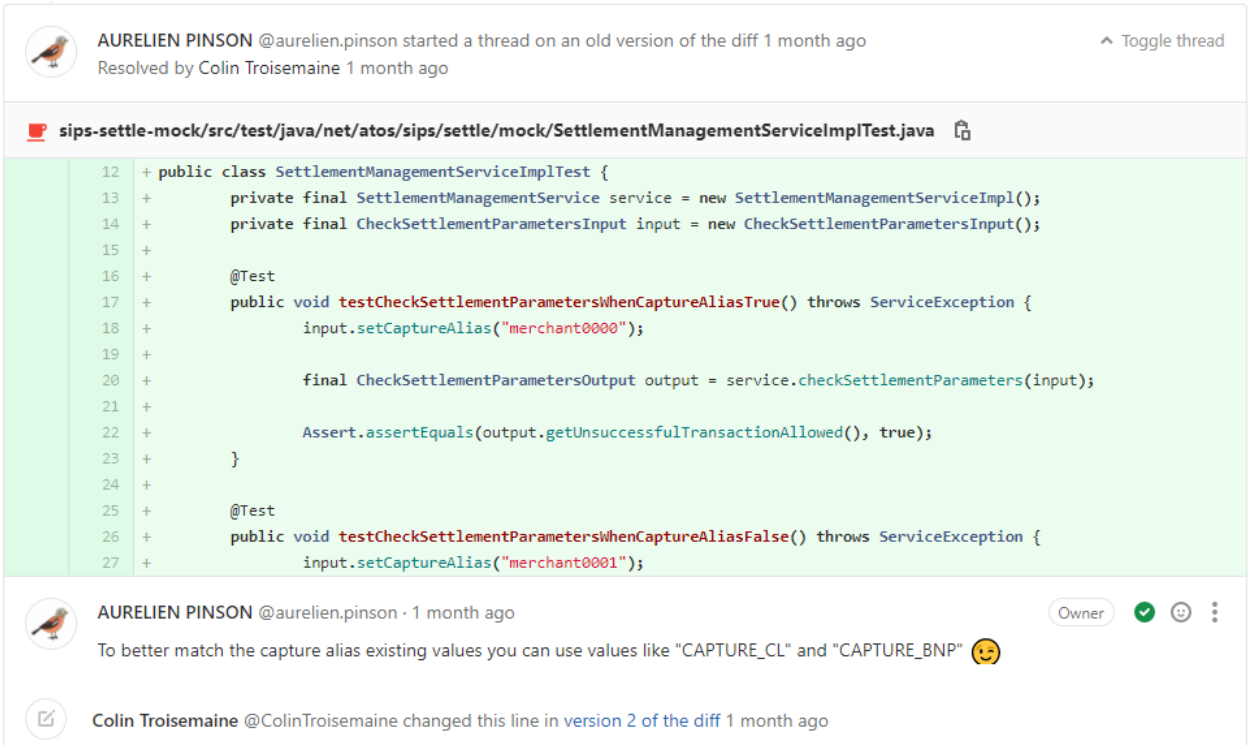

FIGURE 6.1 – Exemple de merge request avec des commentaires

Afin d'améliorer la clarté des messages des *commits* réalisés sur Git, une certaine convention (que j'ai dû respecter) a été adoptée par Worldline :

Type label	Description
feat	Commit is a new feature (or part of feature)
fix	Commit is a bug fix
doc	Commit is a doc update (markdown, javadoc, ...)
style	Commit is a code formatting change
refactor	Commit is a refactoring (no functional change!)
test	Commit adds/fixes some unit or integration tests
chore	Commit is a non code change (pom modification, build process change, ...)
conf	Commit adds/changes application configuration

FIGURE 6.2 – Convention des messages de commit : Un label suivi d'une courte description entre parenthèses puis d'une description plus détaillée

Ce qui donne ce type de messages :

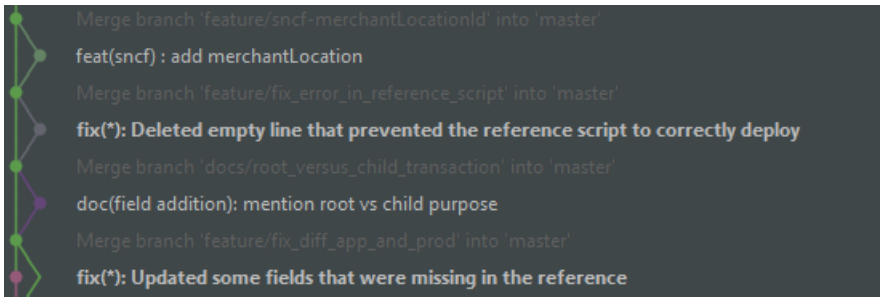

FIGURE 6.3 – Exemple de messages de commit sur le projet bdd-tms

6.1.2 Docker

L'installation de Docker pouvant être compliquée sur certaines machines, Worldline a développé une application nommée *Docker4wl* (comprendre *Docker for Worldline*) qui permet de contourner ces problèmes. L'application Docker est ainsi lancée dans une machine virtuelle hebergée par **VirtualBox** dont les ports sont paramétrés de façon à ce que l'on puisse y accéder depuis notre machine hôte. La fonction de virtualisation de Windows n'étant pas forcément activée de base sur les postes de travail, cette application accélère et facilite la mise en place de Docker.

FIGURE 6.4 – Interface de l'application docker4wl

6.2 Méthode de travail

Ayant réalisé la majeure partie de mon stage en télétravail, la méthode de travail a dû être adaptée. Voici quelques éléments pour illustrer la manière dont mon travail s'effectuait :

6.2.1 Code reviews

Comme je l'ai expliqué dans la section 6.1.1, le code que je produisais était toujours passé en revue. Cela s'est parfois effectué par la fonctionnalité de GitLab qui permet d'ajouter un commentaire à une certaine ligne du code, mais ces revues ont aussi été réalisées en vocal. À ce moment là, la personne qui passait en revue mon code partageait son écran et il lui était ainsi plus facile de commenter et de modifier mon code. C'était aussi l'occasion pour moi de poser toutes mes questions et de discuter de la meilleure approche pour résoudre la tâche que l'on m'avait confiée.

6.2.2 Réunions journalières et bi-hebdomadaire

Tous les jours pendant mon stage, une réunion était planifiée avec Anaëlle Hamon, ma tutrice, pour réaliser un suivi de mon avancement, éviter que je reste bloqué ou que je puisse formuler toutes mes remarques et communiquer mon ressenti sur le stage. Cela me permettait aussi de poser des questions diverses, notamment sur le métier d'ingénieur et les différents rôles au sein d'une équipe de développement.

En plus de ces réunions journalière, un meeting était organisé deux fois par semaine pour que chaque membre de l'équipe sips-core 1 présente son travail depuis la dernière réunion. Cela permettait à chacun d'eventuellement apporter des conseils puisque les compétences sont variées au sein de l'équipe.

6.2.3 Microsoft Lync

Une application qui m'a accompagnée tout au long de mon stage est Microsoft Lync. Elle était mon moyen de communication avec les autres membres de sips-core par messages instantanés, appels ou partages d'écrans. Étant liée à Microsoft Outlook, il était possible d'organiser des meetings ou des réunions en ligne et d'organiser facilement son emploi du temps.

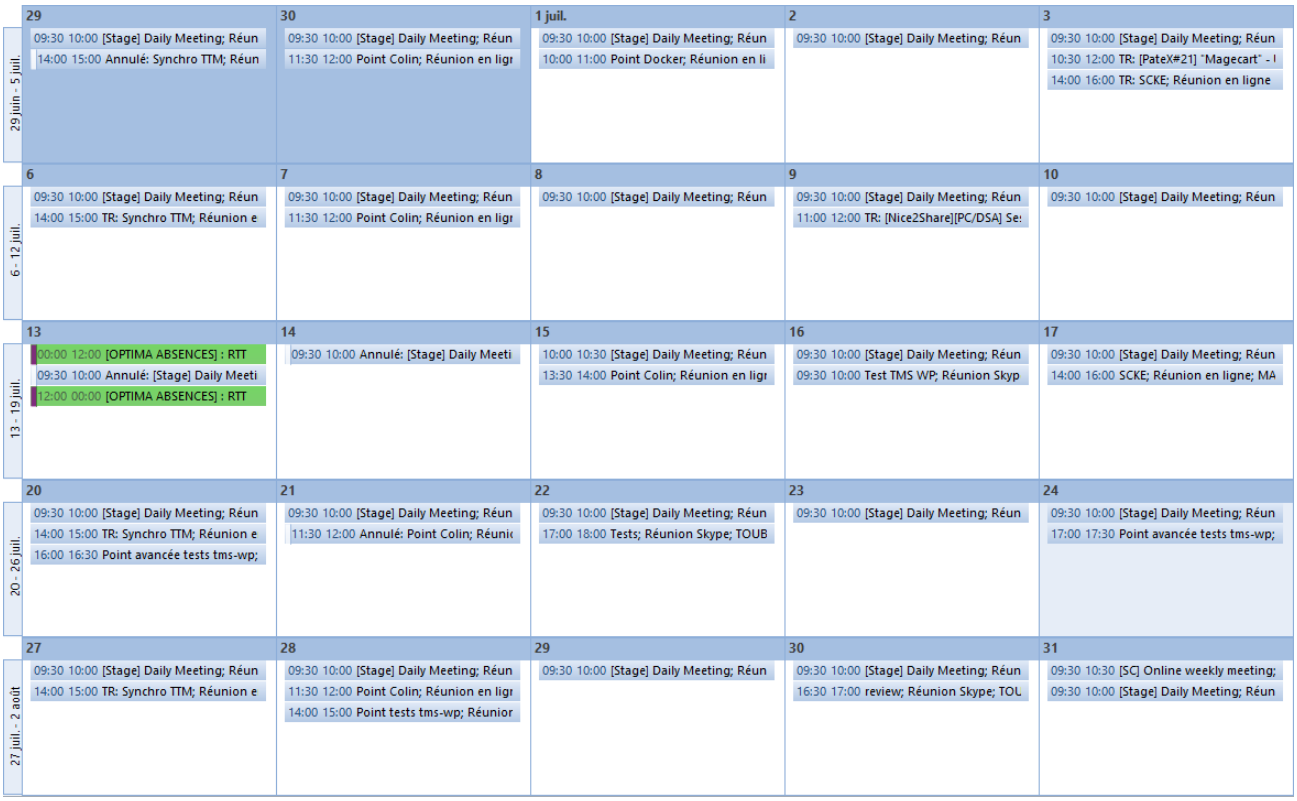

FIGURE 6.5 – Capture d'écran de mon calendrier : Réunions régulières pour que je présente mon travail réalisé, que je pose des questions, etc...

Chapitre 7

Conclusion

Durant ce stage, j'ai eu l'occasion de participer à différents projets qui avaient des objectifs variés. J'ai commencé par travailler à mettre en place une génération de journaux facilement lisibles et formattés en JSON, ce qui m'a permis de construire des algorithmes récursifs dans l'optique de créer des fonctions les plus simples possibles et d'apprendre à mettre en place des tests de façon propre tout en respectant un certain nombre de conventions.

J'ai particulièrement apprécié de travailler sur le second sujet qui portait sur Docker. En effet, il m'a permis de consolider mes connaissances de Docker et de la mise en place de tests. Les technologies que j'ai été amené à utiliser au cours de ce projet ont été très variées. De cette façon j'ai eu le plaisir de relever des défis différents presque tous les jours.

Les interractions avec mes collègues ont toujours été enrichissantes. Ce stage m'a conforté dans l'idée que ce métier me passionnait, mais qu'il me restait encore beaucoup à apprendre !

Docker en action

Rapport de stage 2019 - 2020

Résumé : Ce document a pour but de décrire le travail que j'ai effectué lors de mon stage de quatrième année de parcours d'ingénieur en informatique. Vous y trouverez des explications sur les différents projets et technologies que j'ai été amené à utiliser, ainsi que le détail du travail que j'ai produit.

Mots clé : Worldline, informatique, Docker, Java, Maven, Cucumber

Abstract : The goal of this document is to describe the work a did during my fourth year internship as a computer engineer. You will find explanations on the different projects and technologies that I have been brought to use, as well as the details of the work that I produced.

Keywords : Worldline, computer science, Docker, Java, Maven, Cucumber

Ce document a été formaté selon le format StagePolytech.cls (N. Monmarché, modifié par A. Friot)

École Polytechnique de l'Université de Tours
64 Avenue Jean Portalis, 37200 Tours, France
http://www.polytech.univ-tours.fr