
DOSSIER DE PROJET

TITRE PROFESSIONNEL
CONCEPTEUR DÉVELOPPEUR
D'APPLICATIONS

Csaba Schnitchen

Octobre 2023



Table des matières

Liste des codes sources	3
Table des figures	4
Liste des tableaux	5
1 Liste des compétences du référentiel couvertes par le projet	6
2 Project Summary	7
3 Introduction	8
3.1 Présentation personnelle	8
3.2 Présentation de l'entreprise	8
3.2.1 La petite histoire de SuiviDeFlotte.net	10
4 Gestion de projet	11
4.1 Méthodologie Agile	13
4.2 Jira : L'outil essentiel pour la gestion de projet	16
4.3 Versioning et les environnements	17
4.4 Intégration Continue et Déploiement Continu	22
4.4.1 Le déploiement des paquets de projet Cores	23
4.4.2 Le déploiement du projet Pipeline documentaire	23
5 Cahier des charges	30
5.1 Expression des besoins	30
5.1.1 Concepts généraux	30
5.1.2 Composition de l'outil	30
6 Spécifications fonctionnelles et techniques	33
7 Technologies	37
8 Réalisations	38
8.1 Pipeline Documentaire	38
8.1.1 Installation pour le développement	39
8.1.2 Migrations de la base de données	40
8.1.3 Modèles	42
8.1.4 Le processus d'importation d'un fichier	44
8.2 Mission frontend	52

9 Présentation d'un jeu d'essai	53
10 Veille sur les vulnérabilités de sécurité	54
11 Description d'une situation de travail ayant nécessité des travaux de recherches	55
12 Conclusion	56
13 Remerciements	57
Annexes	58
Annexe A Gestion de projet	59
A.1 Jira : L'outil pour la gestion de projet	59
A.2 Intégration Continue et Déploiement Continu	63

Liste des codes sources

4.1	Le script bash pour installer le projet de Pipeline documentaire en production.	24
4.2	Le fichier <code>docker-compose.yaml</code> du projet Pipeline documentaire utilisé en production.	25
4.3	Le fichier <code>app.dockerfile</code> du projet Pipeline documentaire utilisé en production.	27
8.1	La classe de migration de la table <code>uploads</code>	41
8.2	Le code SQL généré par Laravel sur la base du fichier de migration de la table <code>uploads</code>	42
8.3	Une version simplifiée du modèle <code>Upload</code>	43
8.4	La charge utile (payload) à envoyer contenant le fichier encodé au format base64 et certaines métadonnées associées.	46
8.5	La définition de la route d'importation de fichiers dans le fichier <code>routes/api.php</code>	46
8.6	La classe <code>UploadStoreRequest</code>	47
8.7	La version simplifiée de la classe <code>UploadController</code>	48
8.8	Le constructeur de la classe <code>UploadTrigger</code>	49
8.9	La méthode <code>handle()</code> de la classe <code>UploadTrigger</code>	49
8.10	La méthode <code>recordNewUpload()</code> de la classe <code>UploadTrigger</code>	50
8.11	La méthode <code>saveInDirectories()</code> de la classe <code>UploadTrigger</code>	50
8.12	La classe <code>UploadTriggerLCCC</code>	52
A.1	Le fichier de configuration de Nginx (<code>conf.d</code>) du projet Pipeline documentaire utilisé en production.	63

Table des figures

3.1	Organigramme	9
4.1	La participation des différents comités au processus de traitement des idées d'innovation.	11
4.2	Cérémonies SCRUM.	14
4.3	Programme des sprints chez SuiviDeFlotte. Légende : L, Ma, Me, J, V – jours de la semaine ; DS – Daily Scrum ; SRev – Sprint Review ; Sret – Sprint Retrospective ; SP – Sprint Planning ; ligne pointillée – Sprint précédent ou suivant ; ligne continue – Sprint en cours.	15
4.4	Résumé des relations entre les comités et le processus des sprints.	15
4.5	Diagramme d'activité du traitement des bugs.	16
4.6	Le flux de travail utilisé dans Jira chez SuiviDeFlotte.	17
4.7	Le schéma du versioning et les environnements.	20
4.8	La vue d'ensemble des différents projets chez SuiviDeFlotte et Beepiz. Le projet principal sur lequel j'ai travaillé pendant mon alternance était le projet de Pipeline documentaire, mais j'ai également travaillé sur les projets colorés en gris clair.	21
5.1	Schéma architectural de Pipeline documentaire.	32
6.1	Diagramme d'activité de l'envoi d'une requête POST pour importer un document.	34
6.2	Modèle physique des données du Pipeline documentaire.	35
6.3	Diagramme des classes principales du Pipeline documentaire.	36
7.1	Les principales technologies utilisées dans le cadre du projet.	37
8.1	La structure des dossiers du projet <code>pipelinedoc-docker</code> .	39
8.2	Exemple d'un fichier LCCC.	45
A.1	Le Backlog de produit sous forme de tableau Kanban dans Jira.	59
A.2	Le Backlog de sprint sous forme de tableau Kanban dans Jira.	60
A.3	Un exemple de ticket dans Jira pour le projet d'amélioration de la page d'importation des fichiers des transactions de carburant.	60
A.4	Diagramme d'activité du traitement des idées et des bogues (demandes courantes) entrants chez SuiviDeFlotte.	61
A.5	Diagramme d'activité du traitement des bogues.	62

Liste des tableaux

4.1	Les caractéristiques des différents comités.	12
4.2	Les caractéristiques des différents environnements.	19

1 Liste des compétences du référentiel couvertes par le projet

Compétences professionnelles	Présentation orale	Dossier Projet	Dossier Professionnel
Concevoir et développer des composants d'interface utilisateur en intégrant les recommandations de sécurité			
Maquetter une application	X	X	
Développer une interface utilisateur de type desktop			X
Développer des composants d'accès aux données	X	X	
Développer la partie front-end d'une interface utilisateur web	X	X	
Développer la partie back-end d'une interface utilisateur web	X	X	
Concevoir et développer la persistance des données en intégrant les recommandations de sécurité			
Concevoir une base de données	X	X	
Mettre en place une base de données	X	X	
Développer des composants dans le langage d'une base de données	X	X	
Concevoir et développer une application multicouche répartie en intégrant les recommandations de sécurité			
Collaborer à la gestion d'un projet informatique et à l'organisation de l'environnement de développement	X	X	
Concevoir une application	X	X	
Développer des composants métier	X	X	
Construire une application organisée en couches	X	X	
Développer une application mobile			X
Préparer et exécuter les plans de tests d'une application	X	X	
Préparer et exécuter le déploiement d'une application	X	X	

2 Project Summary

Setipp is a French company, based in Tours, offering telecommunications, isolated worker safety and professional geolocation/fleet management solutions under the brand names Setipp, Beepiz and SuiviDeFlotte.net respectively.

It is within this company that I had the chance to do my work-study training program year to prepare myself for the degree of Application designer and developer.

At the company, I joined the SuiviDeFlotte.net geolocation web development team. My tutor and I, along with a colleague who joined us later, we worked on a completely new project, an API, called « Document Pipeline ». The purpose of the API was to process various types of documents coming from other services of the company, such as text files containing fuel purchase transactions or scanned invoices. It needed to extract and store the data from these documents in the database. The API had to be designed in a way that it could easily be expanded to handle processing of new file types in the future.

We developed the API using the Laravel framework and stored the data in several MariaDB databases. During the project, I also worked on transforming the web user interface for uploading files containing fuel purchase transactions within the geolocation/fleet management website, another Laravel project, to adapt its functionality to the new API. This interface utilized the Blade templating language and the Vue.js framework.

Within the geolocation development team, we followed agile principles, specifically applying the SCRUM methodology to organize our work, which allowed us to adapt flexibly to emerging needs.

3 Introduction

3.1 Présentation personnelle

Je m'appelle Csaba SCHNITCHEN, j'ai 46 ans. Je suis né en Hongrie, où j'ai obtenu un diplôme en biologie/écologie en 2002, suivi d'un doctorat en sciences de l'environnement en 2007. Jusqu'en 2020, j'ai travaillé en tant que professeur et chercheur dans deux universités. Au cours de mes recherches en écologie, j'ai découvert la programmation et j'ai utilisé les langages R et Python pour créer différents modèles écologiques et des graphiques pour des publications scientifiques. Par la suite, j'ai appliqué mes connaissances en Python pour créer un site web universitaire en utilisant le framework Django.

En 2020, j'ai déménagé définitivement en France. Ici, j'ai pris la décision de changer de profession et de me lancer dans le domaine de l'informatique. En 2022, j'ai suivi la formation Développeur web et web mobile au CEFIM, ce qui a considérablement approfondi mes connaissances en informatique et en programmation, acquises en autodidacte. Par la suite, j'ai poursuivi mes études en suivant la formation Concepteur développeur d'applications, toujours au CEFIM. Cette formation se déroulait en alternance. Pour cela, j'ai eu la chance de rejoindre Setipp, une société française, basée à Tours, proposant des solutions de télécommunications, de sécurité des travailleurs isolés et de géolocalisation professionnelle/gestion de flotte, sur un rythme de trois semaines en entreprise et une semaine CEFIM.

3.2 Présentation de l'entreprise

Setipp est une société française dont le siège est à Tours et qui possède également des bureaux à Lille, Paris, Strasbourg, Nantes, Lyon, Toulouse et Montpellier.

Setipp propose trois familles de services et de produits à ses clients sous trois marques : Setipp, Beepiz et SuiviDeFlotte.net. Setipp fournit des services de télécommunications aux entreprises. Beepiz propose des applications web et mobiles qui permettent aux travailleurs isolés de se protéger en toutes circonstances et d'accéder aux services d'urgence. SuiviDeFlotte.net propose des solutions professionnelles de géolocalisation et de gestion de parc. Leurs services en ligne incluent des fonctions telles qu'aider les conducteurs à économiser du carburant et aider les entreprises à décider quels véhicules valent la peine de passer à l'électrique.

Au sein de l'entreprise, j'ai intégré l'équipe de développement de la géolocalisation de SuiviDeFlotte.net tel qu'il apparaît dans l'organigramme de SuiviDeFlotte (Figure 3.1).

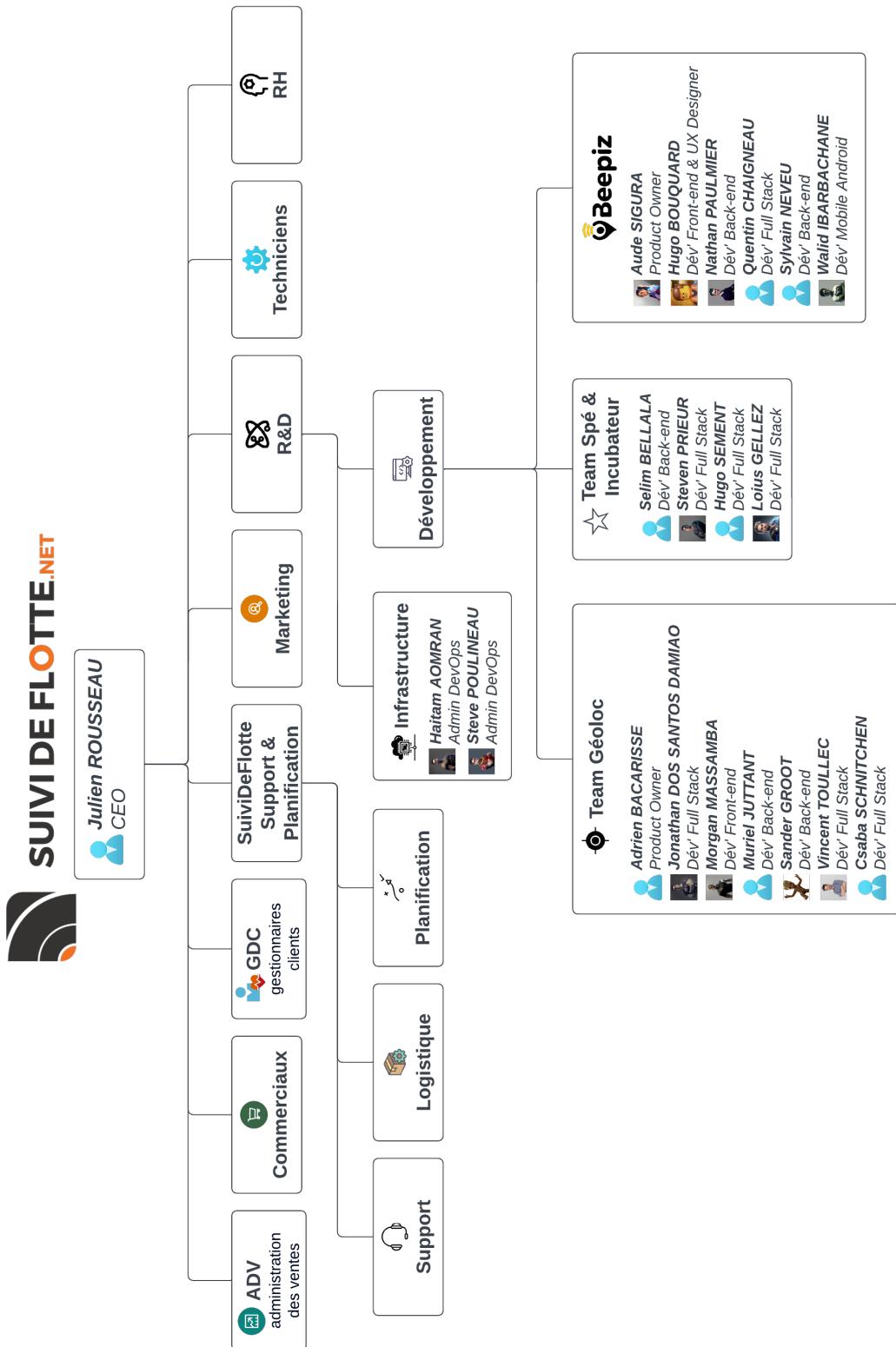


FIGURE 3.1 – Organigramme

3.2.1 La petite histoire de SuiviDeFlotte.net

SuiviDeFlotte.net, créée en 2001, est la branche de l'entreprise spécialisée dans la géolocalisation et la gestion de flottes de véhicules et d'objets connectés pour les entreprises. À l'origine de cette initiative se trouve Julien Rousseau, actuel Président Directeur Général, qui a remarqué que la gestion de la flotte automobile était le deuxième poste de dépense le plus important pour les entreprises. Pour optimiser ces parcs automobiles et améliorer la productivité des métiers nécessitant des interventions sur le terrain, il a envisagé d'appliquer les succès de la télécommunication aux véhicules.

Les bénéfices potentiels de la télématicque embarquée se sont avérés concluants, conduisant ainsi à la création officielle de SuiviDeFlotte.net, pionnière de la télématicque embarquée en France. Initialement centrée sur la géolocalisation des véhicules, l'entreprise a progressivement élargi ses services pour offrir des outils complets de gestion de flottes, incluant la géolocalisation, la gestion de parc et l'éco-conduite. Depuis ses débuts, SuiviDeFlotte.net propose ses services via une plateforme SaaS, permettant d'ajouter de nouvelles fonctionnalités aux utilisateurs sans installation ni maintenance.

Aujourd'hui, l'entreprise se focalise sur l'innovation, cherchant à répondre aux besoins de ses utilisateurs en proposant trois grandes mises à jour par an, intégrant plus de 100 nouvelles fonctionnalités chaque année. SuiviDeFlotte.net conçoit et commercialise des solutions clés en main de géolocalisation, écoconduite et gestion de flottes de véhicules (VL, VU, poids lourds), qui sont utilisées par 4000 entreprises, qu'il s'agisse de TPE, PME ou entités de grands groupes. Elle compte 50 collaborateurs, génère un chiffre d'affaires de 7 millions d'euros et consacre 25% de son effectif à la recherche et développement.

4 Gestión de proyecto

Pendant mon alternance, j'ai très vite compris que pour l'entreprise, il est important de maintenir son esprit innovant, de constamment générer davantage de valeur, et d'être à l'écoute tout en s'ajustant selon les exigences de sa clientèle. La politique d'innovation de l'entreprise repose sur les recommandations émanant à la fois de ses clients et de ses collaborateurs. Un comité dédié à l'innovation se rassemble hebdomadairement pour examiner les suggestions les plus récentes. Chaque concept est traité, trié et priorisé. Il s'agit d'un processus en plusieurs étapes qui implique également d'autres comités, comme l'illustre la Figure 4.1 et la Table 4.1. Par la suite, l'ensemble de ces propositions est transmis au département Recherche et Développement en vue de la création de nouvelles fonctionnalités. Tous les quatre mois, de nouvelles options viennent enrichir l'ensemble des services, désignées sous le terme « Editions ».

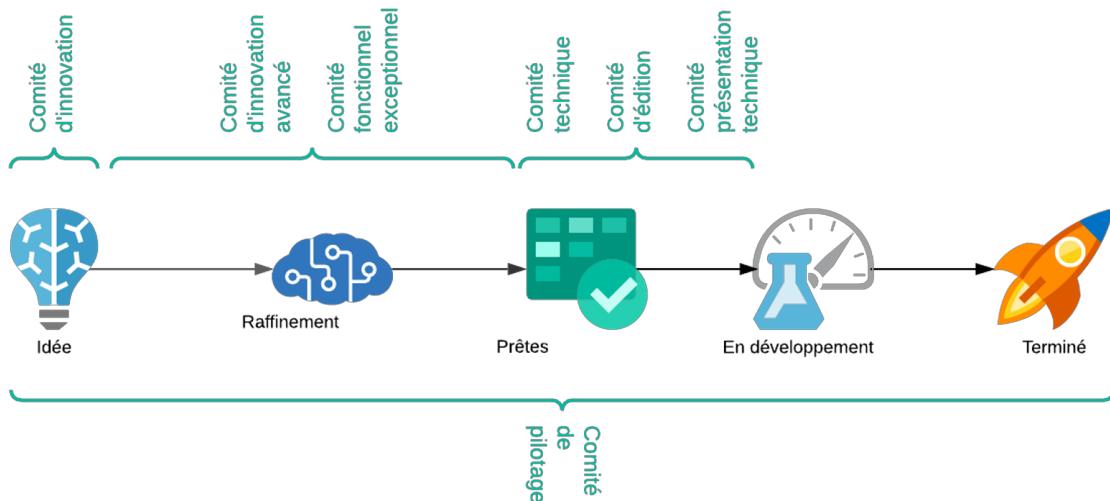


FIGURE 4.1 – La participation des différents comités au processus de traitement des idées d'innovation.

Une édition représente le fruit de quatre mois de travail de développement, cependant, son élaboration ne s'arrête pas là. Elle englobe la mise en place et la planification des activités de communication (marketing), la formation des équipes support et commerce, la rédaction de manuels et de tutoriels destinés aux clients, ainsi que la préparation des prochaines éditions à venir.

Table 4.1: Les caractéristiques des différents comités.

Nom du comité	Récur-rence	Objectif	Entrées	Sorties
Comité d'innovation	Hebdo-madaire	Première analyse des idées exprimées puis étude de l'intérêt de chaque idée.	Liste des idées	Idées clarifiées, note par niveau d'intérêt
Comité d'innovation avancé	Mensuel	Clarifier les spécifications fonctionnelles des idées exprimées au cours des comités d'innovations.	Ordre du jour des idées, déjà analysées	Réponses aux questions à l'ordre du jour, ajoutées aux backlog
Comité fonctionnel exceptionnel	Excep-tionnel	Le PO sollicite une partie prenante identifiée, qui devrait lui permettre de lever un certain nombre de questions autour d'une fonctionnalité.	Fonctionnalités, déjà analysées et comportant toujours des questions	Les réponses sont ajoutées aux fonctionnalités
Comité tech	En fonction des items en attente de chiffrage dans le backlog. Au moins hebdo-madaire.	Une présentation des items, validés fonctionnellement, pour faire ressortir un macro-chiffrage, réalisé par l'équipe d'expert.	Fonctionnalités prêtes du backlog produit	Macro-chiffrage ou questions fonctionnelles
Comité pilotage	Bimestriel	Donner une vision claire de l'avancement du travail réalisé pour les services concernés.	Indicateur clé de performance à partager	Adaptations à mettre en œuvre
Comité d'édition	Bimestriel	Établir une constitution d'édition à partir des idées prêtes. En déduire un objectif d'édition permettant de fédérer autour d'une réalisation.	Fonctionnalités prêtes, analyse du temps disponible faite à partir des paramètres vitesse / nombre de demandes courantes / dette technique ...	Liste des composants d'édition, objectif d'édition
Comité présentation technique / Kickoff Edition	En début d'édition	Une présentation des items embarqués dans l'édition, ainsi qu'un focus sur l'objectif d'édition.	L'objectif, le contenu d'édition	Le retour de l'équipe sur le contenu et l'objectif

4.1 Méthodologie Agile

L'équipe de développement de la géolocalisation de SuiviDeFlotte – comme les autres équipes de développement de l'entreprise – travaille selon la méthodologie agile SCRUM. Cette méthodologie est une approche de gestion de projet qui met l'accent sur la flexibilité, la collaboration et la livraison continue. Elle est largement utilisée dans le développement de logiciels et peut également être appliquée à d'autres domaines. SCRUM divise un projet en cycles appelés « itérations » ou « sprints » de courte durée, généralement de deux à quatre semaines, pendant lesquels une partie du travail est accomplie et livrée.

Les termes clés de la méthodologie SCRUM sont énumérés ci-dessous et illustrés dans la Figure 4.2.

Product Owner (Propriétaire du Produit) La personne responsable de définir et de prioriser les éléments du produit à développer. Le Propriétaire du Produit représente les besoins des utilisateurs et des parties prenantes.

Scrum Master (Maître de Scrum) Le facilitateur du processus SCRUM. Le Scrum Master s'assure que l'équipe suit les principes SCRUM, élimine les obstacles et favorise un environnement de travail efficace.

Équipe de Développement Le groupe de professionnels chargé de concevoir, développer, tester et livrer les éléments du produit à la fin de chaque sprint.

User Story (Histoire Utilisateur) Une Histoire Utilisateur est une courte description d'une fonctionnalité ou d'un aspect du produit, racontée du point de vue de l'utilisateur. Elle suit généralement le format « En tant que [utilisateur], je veux [action] afin de [objectif] ». Les Histoires Utilisateurs sont des éléments du Carnet de Produit et aident à définir les fonctionnalités du produit du point de vue de l'utilisateur.

Story Point (Point d'Histoire) Le Point d'Histoire est une unité relative utilisée pour estimer la complexité, l'effort et la taille des Histoires Utilisateurs ou des tâches de développement. Il n'a pas de valeur absolue, mais il sert à comparer la difficulté relative entre différentes Histoires Utilisateurs. Les équipes de développement attribuent des points d'histoire lors des estimations, ce qui les aide à planifier la quantité de travail qu'elles peuvent accomplir dans un sprint donné.

Epic (Épique) Un Épic est une unité de travail plus large que les Histoires Utilisateurs individuelles. Il représente généralement un ensemble de fonctionnalités, de tâches ou de travaux qui sont trop importants pour être traités dans un seul sprint. Les Épics sont souvent des objectifs à long terme qui sont décomposés en Histoires Utilisateurs plus petites et gérables. Ils aident à organiser et à structurer le développement du produit en regroupant des éléments liés autour d'un thème ou d'un objectif commun. Les Épics sont inclus dans le Carnet de Produit et sont priorisés en fonction de leur valeur pour l'utilisateur et du contexte global du projet.

SCRUM encourage la transparence, l'adaptabilité et la collaboration continue entre les membres de l'équipe et les parties prenantes, ce qui permet de s'adapter aux changements et de fournir rapidement de la valeur tout au long du projet.

Dans l'équipe Géoloc, le processus suit un calendrier de sprints de deux semaines (Figure 4.3). Chaque cycle commence par un ensemble de réunions clés qui

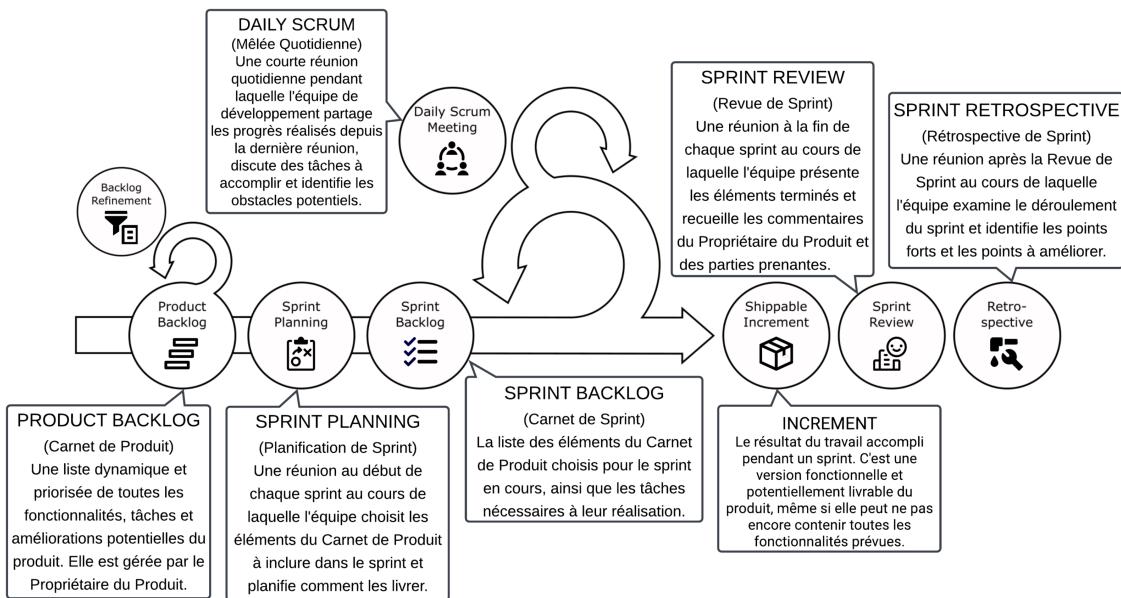


FIGURE 4.2 – Cérémonies SCRUM.

ont lieu tous les deuxièmes mardis. Cette journée englobe la Revue de Sprint, la Rétrospective de Sprint et la Planification de Sprint. Lors de la Revue de Sprint, les éléments achevés sont présentés au Propriétaire du Produit et aux parties prenantes, les retours sont recueillis et les priorités sont ajustées si nécessaire. La Rétrospective de Sprint offre l'opportunité à l'équipe de réfléchir aux succès et d'identifier les domaines à améliorer, favorisant une culture d'amélioration continue. Ensuite, la Planification de Sprint implique la sélection des Histoires Utilisateurs du Carnet de Produit à inclure dans le sprint à venir, en tenant compte de leur complexité et de leur priorité. Pour l'estimation de la complexité des tâches, des Points d'Histoire basés sur la séquence de Fibonacci (1, 2, 3, 5, 8, 13) sont utilisés. Cette approche aide à attribuer des valeurs relatives à différentes tâches et assure une évaluation cohérente de la charge de travail.

À la fin de chaque sprint, les nouveaux incrémentés sont également mis en production. Grâce à l'approche Agile et à la fréquence des mises en production, les modifications apportées à l'application sont rapidement portées à la connaissance de l'utilisateur.

Chaque jour ouvrable débute par une Mélée Quotidienne de 15 minutes, au cours de laquelle les progrès depuis la dernière réunion sont passés en revue, les tâches sont discutées et les obstacles potentiels sont identifiés. Ces réunions se déroulent généralement en ligne via Google Meet pour accueillir les collègues travaillant à distance et garantir la participation de tous les membres de l'équipe, peu importe leur emplacement. Par contre, pour maintenir la communication et la cohésion de l'équipe, les réunions du mardi de chaque deuxième semaine sont des réunions en personne. Cette pratique encourage les échanges directs, la collaboration étroite entre les membres de l'équipe et facilite la résolution rapide de tout problème ou obstacle pouvant survenir.

Actuellement, les rôles du Propriétaire du Produit et du Maître de Scrum ont été temporairement fusionnés en une seule personne. Cette configuration permet au Propriétaire du Produit de gérer les responsabilités du projet tout en facilitant

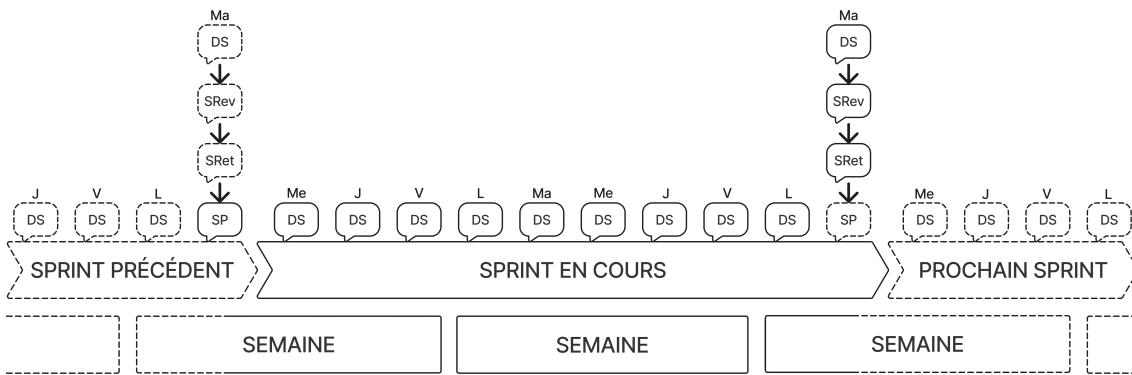


FIGURE 4.3 – Programme des sprints chez SuiviDeFlotte. Légende : L, Ma, Me, J, V – jours de la semaine ; DS – Daily Scrum ; SRev – Sprint Review ; Sret – Sprint Retrospective ; SP – Sprint Planning ; ligne pointillée – Sprint précédent ou suivant ; ligne continue – Sprint en cours.

le processus SCRUM, en maintenant une communication fluide avec l'équipe de développement.

En complément des cycles de deux semaines des Sprints, l'opération de l'entreprise comprend également un cycle plus long. En effet, comme je l'ai mentionné dans l'introduction de ce chapitre, l'entreprise publie une nouvelle version, appelée nouvelle Edition, de ses services en ligne tous les quatre mois, soit trois fois par an. Cela implique qu'il y a environ huit sprints pour que les développeurs puissent concevoir les nouvelles fonctionnalités des nouvelles éditions.

La Figure 4.4 résume les relations entre les comités et le processus des sprints. Comme on peut le voir, les tickets du Carnet de produit proviennent du Comité d'innovation, du Comité d'innovation avancée, du Comité fonctionnel exceptionnel, du Comité d'édition et du Comité d'innovation technique. Ensuite, le Comité d'édition sélectionne les tickets pour la prochaine Edition et le Comité de présentation technique les présente à l'équipe de développeurs. Une autre source de tickets dans le Backlog de Produit est l'équipe de support qui crée des tickets de bogues. Le traitement des bogues est illustré par la Figure 4.5. La section suivante explique comment ces tickets sont traités dans un outil appelé Jira.

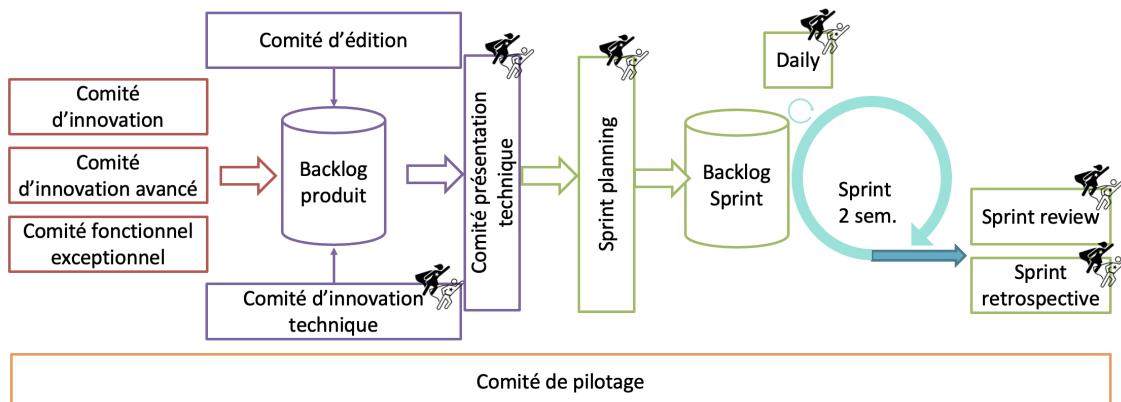


FIGURE 4.4 – Résumé des relations entre les comités et le processus des sprints.

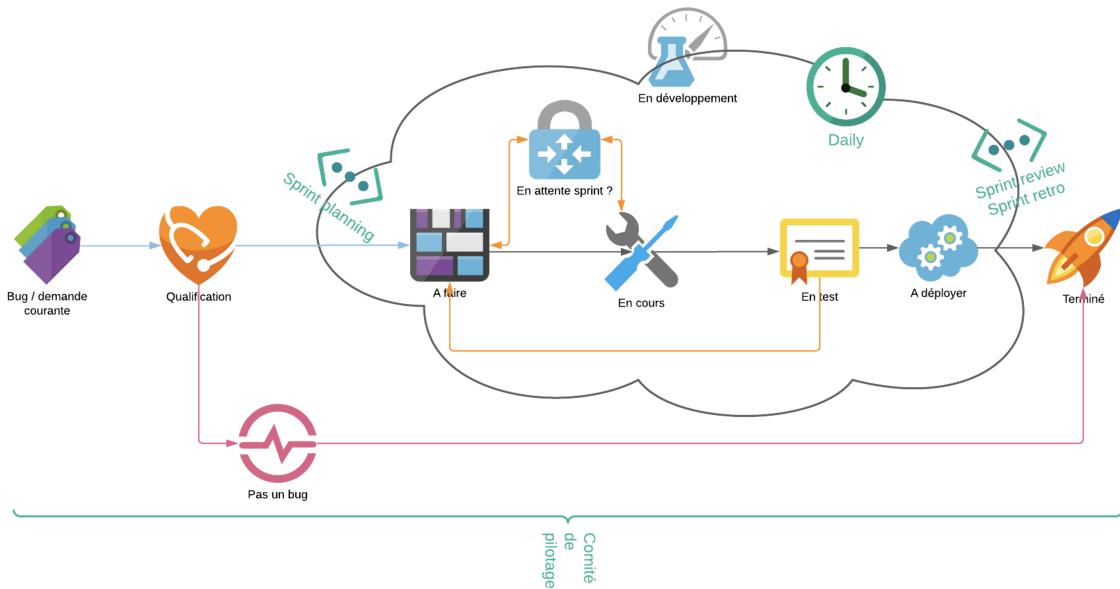


FIGURE 4.5 – Diagramme d'activité du traitement des bugs.

4.2 Jira : L'outil essentiel pour la gestion quotidienne de projet

Au sein de l'entreprise, les équipes SCRUM se servent de Jira afin de consigner et de suivre tous les aspects de leur travail.

Jira est un outil essentiel pour les équipes SCRUM car il facilite la gestion complète du processus de développement agile. Il permet aux équipes de collaborer de manière efficace et de suivre chaque étape du cycle de vie du projet. Avec Jira, les équipes SCRUM peuvent créer, organiser et hiérarchiser leur Carnet de Produit en ajoutant des Histoires Utilisateurs, des tâches, des Épics et des bugs.

L'outil facilite la planification des sprints en permettant aux équipes de sélectionner les éléments du Carnet de Produit à inclure dans chaque itération. Les équipes peuvent estimer la complexité des tâches à l'aide de Story Points et suivre leur progression au fil du temps.

Les fonctionnalités de suivi des tâches et des problèmes dans Jira aident les équipes à gérer leur travail quotidien. Chaque membre de l'équipe peut mettre à jour l'état de ses tâches, signaler les obstacles et collaborer de manière transparente avec les autres membres de l'équipe.

Les réunions SCRUM telles que la Mêlée Quotidienne, la Revue de Sprint et la Rétrospective de Sprint peuvent être orchestrées efficacement grâce à Jira. L'outil permet de suivre les progrès, de partager les résultats et de documenter les réflexions pour chaque itération.

Les flux de travail (workflows) dans Jira constituent un mécanisme central pour orchestrer et suivre le déroulement des tâches et des projets. Ces flux définissent les étapes spécifiques à suivre pour qu'une tâche ou un problème progresse, de sa création à son achèvement. Les équipes peuvent personnaliser ces flux pour refléter leurs processus uniques, en définissant les transitions entre les étapes et en assignant des responsabilités à différents membres de l'équipe. Les flux de travail de Jira

contribuent à maintenir la transparence, à améliorer l'efficacité et à garantir que toutes les parties prenantes restent informées de l'avancement du travail. Grâce à cette fonctionnalité, les équipes peuvent gérer avec agilité et précision les tâches, tout en favorisant une collaboration fluide et une visibilité accrue sur les projets.

Chez SuiviDeFlotte, le diagramme d'activité présenté dans les Annexes, dans la Figure A.4 (page 61) est la base du flux de travail dans Jira. La partie supérieure gauche du diagramme illustre à nouveau le traitement des idées novatrices par les différents comités. En revanche, la partie supérieure droite présente le traitement des bogues (demandes courantes) qui proviennent de l'équipe de support. Ensuite, le propriétaire du produit sélectionne les tickets pour le prochain sprint à partir de l'ensemble des tickets prêts. Ce diagramme contient les étapes suivantes, qui sont utilisées comme étapes du flux de travail dans Jira : en raffinement, à faire, en cours, en test, à déployer, terminé, abandonné.

Un autre diagramme d'activité montre plus en détail le traitement des bogues (Annexes, Figure A.5, page 62). À partir de ce diagramme, nous pouvons ajouter quelques étapes supplémentaires à la liste des étapes du flux de travail : en attente, en attente support.

Le diagramme de flux de travail qui en résulte et qui est actuellement utilisé est présenté à la Figure 4.6.

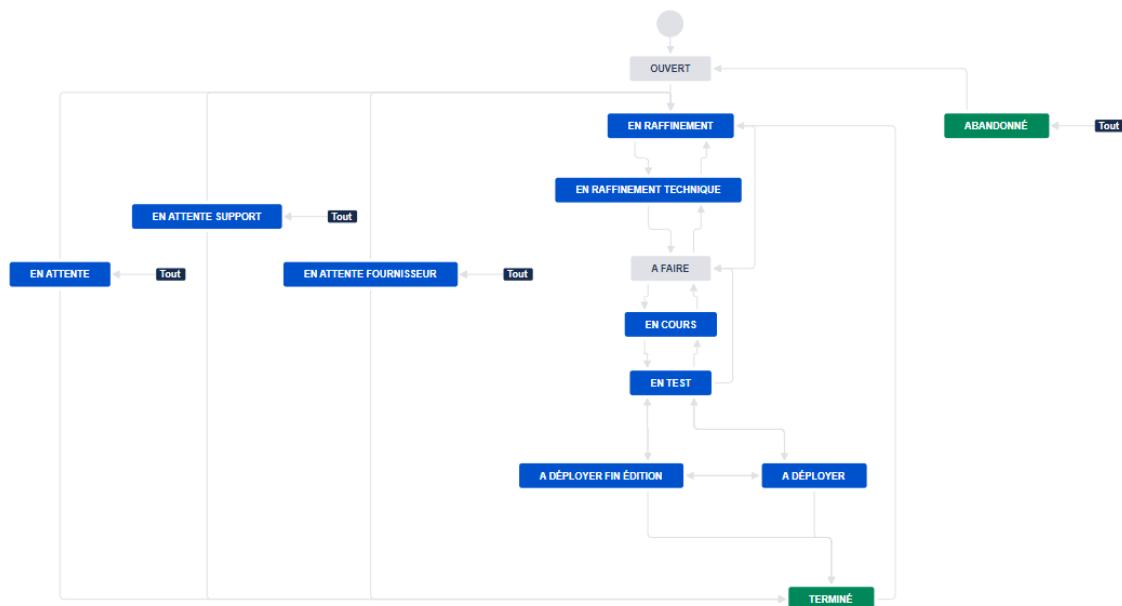


FIGURE 4.6 – Le flux de travail utilisé dans Jira chez SuiviDeFlotte.

Dans les Annexes, la Figure A.1 (page 59) et la Figure A.2 (page 60) présentent un exemple de Backlog de produit et de Backlog de sprint sous forme de tableau Kanban, et la Figure A.3 (page 60) donne un exemple d'un ticket.

4.3 Versioning et les environnements

Dans l'entreprise, les équipes de développement utilisent Git pour la gestion des versions et GitLab pour gérer le processus de développement.

Git est un système de gestion de version décentralisé largement utilisé dans le développement de logiciels. Il permet aux équipes de collaborer efficacement sur des

projets en suivant les modifications apportées aux fichiers au fil du temps. Grâce à Git, les développeurs peuvent créer des branches pour travailler sur des fonctionnalités spécifiques ou des corrections de bugs sans perturber le code principal. Les commits, qui représentent des enregistrements de changements, sont la pierre angulaire de Git, permettant de garder une trace claire de l'évolution du code.

GitLab, quant à lui, est une plateforme de gestion de développement logiciel basée sur Git. Elle offre un environnement complet pour le cycle de vie du développement, de la planification à la surveillance. GitLab permet aux équipes de suivre les problèmes, de planifier les sprints, de gérer les demandes d'extraction et de créer des pipelines d'intégration continue pour automatiser les tests et le déploiement. En regroupant toutes ces fonctionnalités au même endroit, GitLab facilite la collaboration entre les membres de l'équipe et permet une gestion transparente et efficace des projets de développement.

Parmi ces nombreuses fonctionnalités, les équipes de DevOps et de développement n'utilisent cependant pas les fonctionnalités de gestion de projet et d'intégration continue/de livraison continue de GitLab, puisqu'elles utilisent Jira (comme nous l'avons vu dans la section précédente) et TeamCity (comme nous le verrons dans la section suivante) à ces fins. GitLab est donc principalement utilisé comme un hébergeur de dépôt de code et un outil de révision de code avec des fonctionnalités telles que les demandes de fusion (merge request).

La Figure 4.7 illustre les principes du versioning et les différents environnements utilisés pour héberger les différentes versions du code. La Table 4.2 apporte quelques compléments d'information.

Du point de vue des versions et des environnements, le processus de développement entre deux éditions est le suivant. Avant la sortie publique de la nouvelle édition, il y a toujours une sortie interne. La sortie interne est créée par la fusion de la branche `develop` dans la branche `master` avec la balise `{numéro d'édition}.0`. Après cela, une nouvelle branche de `release` est créée à partir de la branche `master` avec le nom `release/{numéro d'édition}`. Cette nouvelle branche `release` n'est pas touchée avant la sortie publique.

Entre la sortie interne et la sortie publique, des correctifs urgents (`hotfix`) peuvent être apportés. Lorsque le développeur travaille sur un `hotfix`, il crée d'abord une branche à partir de l'ancienne branche `release` avec un nom `hotfix/{bogue}`, il y dépose ses modifications en travaillant dans son environnement sandbox développeur, puis il crée une demande de fusion pour réintégrer cette branche dans l'ancienne branche `release`. Lorsqu'une décision positive est prise concernant la sortie publique, l'ancienne branche `release` est fusionnée dans les branches `master` et `develop`, l'ancienne branche `release` est clôturée (archivée), la branche `master` est fusionnée dans la nouvelle branche `release`, puis la nouvelle branche `release` est fusionnée dans la branche `develop`, enfin la branche `master` est mise en production dans l'environnement de production avec la balise `{numéro d'édition}.1`. Pendant la période entre la sortie interne et la sortie publique, les développeurs peuvent travailler sur de nouvelles fonctionnalités, mais celles-ci ne seront pas intégrées dans l'édition actuelle, elles ne pourront l'être que dans la prochaine.

Après la sortie publique de la nouvelle édition, les développeurs commencent à travailler sur la prochaine édition. Lorsqu'ils travaillent sur des bogues, des correctifs (`fix`), ils créent une branche à partir de la branche `release` avec un nom `fix/{bogue}`, ils travaillent dans leur environnement sandbox, et lorsqu'ils ont terminé, ils

créent une demande de fusion pour fusionner leur branche dans la branche `releas`_j`se`. Lorsqu'ils travaillent sur des fonctionnalités, cela se passe de la même manière, sauf qu'ils créent leur branche à partir de la branche `develop` avec un nom `feat/`_j`{fonctionnalité}` et bien sûr ils créent la demande de fusion pour fusionner leur branche dans la branche `develop`. Au cours de cette période, les corrections peuvent être fusionnées dans les branches `master` et `develop` et elles peuvent être mises en production si nécessaire à la fin des sprints ou même au milieu de ceux-ci. Dans ce cas, une nouvelle subversion est ajoutée au commit de fusion sous la forme d'une balise (`{numéro d'édition}.{numéro d'increment}`).

Table 4.2: Les caractéristiques des différents environnements.

Environnement	Pour quels services	Branches	Type de comits	Déploiement TeamCity
sandbox développeur	Développeur	<code>feat/</code> <code>{fonctionnalité}</code> <code>fix/{bogue}</code> <code>hotfix/{bogue}</code>		—
recette-develop	Le PO vérifie les fonctionnalités	<code>develop</code>	<code>feat</code>	Auto
recette-releases	Le PO vérifie les corrections	<code>releases/édition</code>	<code>fix, hotfix</code>	Auto
recette-master	Autres services pour test (direction, marketing, commerce ...)	<code>master</code>	(env juste avant la prod, préproduction)	Manuel, section master dans TeamCity, demande via ticket MEP à DevOps
production		<code>tag</code>	Commit en production avec tag <code>{numéro d'édition}</code> . <code>{numéro d'increment}</code>	Manuel, section production dans TeamCity, demande via ticket MEP à DevOps

A ce stade, il est important de noter que le processus expliqué ci-dessus ne concerne pas qu'un seul projet, mais que, puisqu'il y a plusieurs projets chez SuiviDeFlotte, il les concerne tous. Cela signifie que chaque projet dispose de ses propres branches de `develop`, de `release` et `master`, ainsi que de ses propres environnements. Pour donner une vue d'ensemble des différents projets, la Figure 4.8 présente une image de l'architecture de projets chez SuiviDeFlotte et Beepiz.

Le nouveau projet sur lequel j'ai travaillé la plupart du temps pendant mon alternance est le projet Pipeline documentaire. Cependant, j'ai reçu de temps en temps des tickets de bogues provenant des projets Portail, Gestion de parc, API et Cores. De plus, j'ai travaillé sur une mission d'amélioration du frontend dans le projet Gestion de parc, qui était lié au projet Pipeline documentaire.

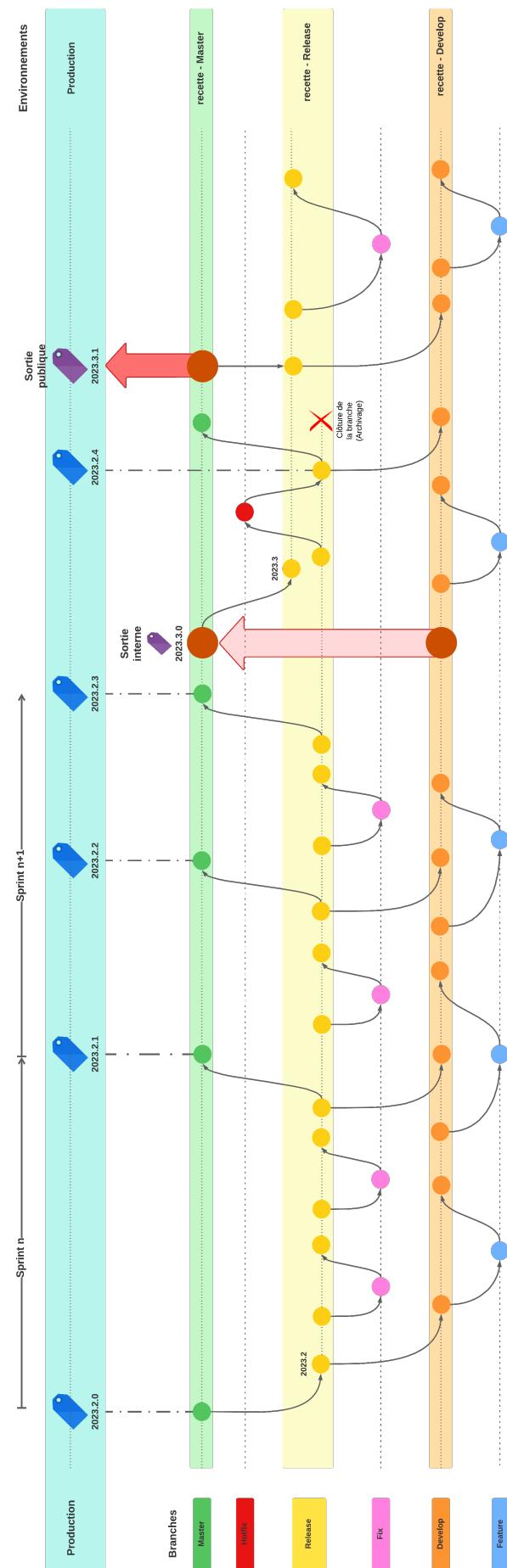


FIGURE 4.7 – Le schéma du versioning et les environnements.

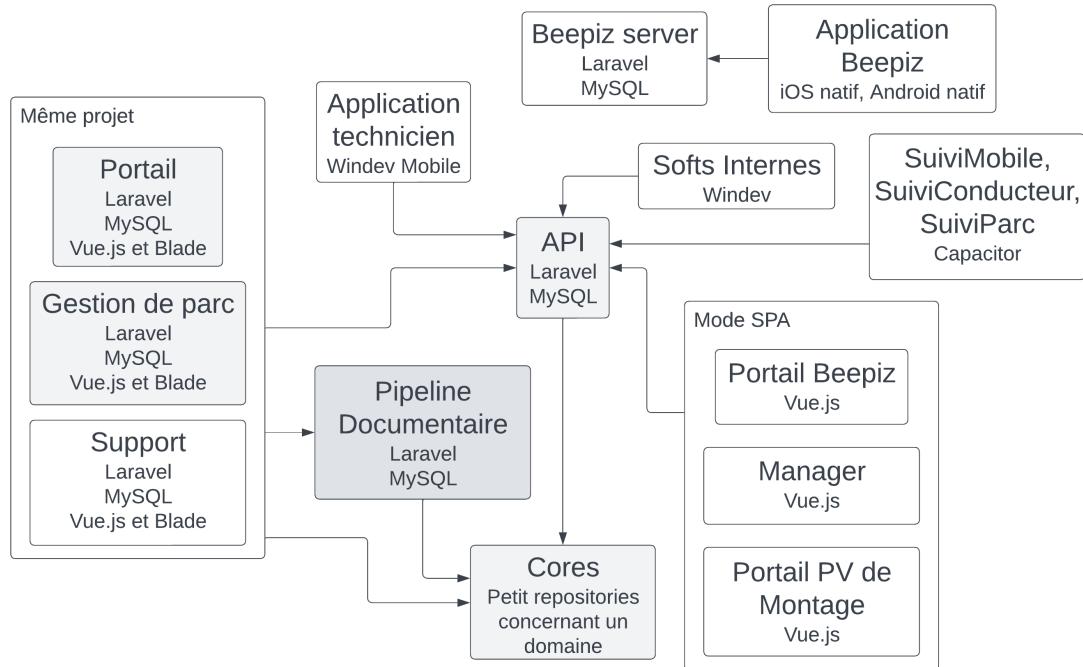


FIGURE 4.8 – La vue d’ensemble des différents projets chez SuiviDeFlotte et Beepiz. Le projet principal sur lequel j’ai travaillé pendant mon alternance était le projet de Pipeline documentaire, mais j’ai également travaillé sur les projets colorés en gris clair.

Le Portail, la Gestion du parc et le Support font partie du même projet Laravel qui est une application SaaS disponible pour les clients de SuiviDeFlotte. La partie Portail fournit les fonctionnalités de géolocalisation, la partie Gestion de parc fournit les fonctionnalités de gestion de parc comme son nom l’indique et la partie Support aide l’équipe de support dans son travail. L’API est un projet Laravel backend central qui fournit des données à d’autres projets à partir de la base de données. Le projet Cores implique un grand nombre de paquets PHP relativement petits qui sont stockés dans un dépôt de paquets privé créé par Satis.¹ Ces paquets contiennent des fonctionnalités communes qui peuvent être utiles dans plusieurs autres projets. Cependant, lorsque nous avons commencé à travailler sur le projet de Pipeline documentaire, nous avons commencé à travailler sur leur refactorisation et au cours de ce processus, nous avons créé un nouveau paquet (`core-redesing-share`) avec du code refactorisé provenant d’autres paquets et qui n’est pas encore utilisé, mais seulement par le Pipeline documentaire pour le moment.

1. Satis est un outil qui permet aux développeurs PHP de créer un dépôt de paquets privé pour les dépendances de leurs projets. Il offre un contrôle accru sur la distribution des paquets, une sécurité améliorée et des installations de paquets plus rapides, en créant un registre Composer statique qui peut être hébergé n’importe où (même via Docker, localement). Disponible sur <https://github.com/composer/satis>

4.4 Intégration Continue et Déploiement Continu

L'équipe Infrastructure/DevOps utilise TeamCity qui est une plateforme d'intégration continue et de livraison continue largement adoptée dans le domaine du développement logiciel. Elle permet aux équipes de développeurs de automatiser le processus de construction, de test et de déploiement de leurs applications. TeamCity offre un environnement convivial où les développeurs peuvent configurer des pipelines d'intégration continue en spécifiant les étapes nécessaires, telles que la compilation du code, les tests unitaires et les déploiements sur différents environnements.

Une caractéristique clé de TeamCity est sa capacité à détecter automatiquement les changements dans le code source et à déclencher des builds et des tests en conséquence. Cela permet aux équipes de détecter rapidement les problèmes et de s'assurer que le code reste stable et fonctionnel à chaque étape du développement.

En outre, TeamCity offre des fonctionnalités avancées telles que la gestion des agents de construction, la parallélisation des tâches, la gestion des paramètres de build et des rapports détaillés sur les résultats des tests. Cette plateforme aide les équipes à maintenir un processus de développement fluide et à garantir la qualité du code grâce à l'automatisation et à la surveillance continue.

Comme l'indique la Table 4.2 (page 19), le déploiement des branches `develop` et `release` dans les environnements `recette-develop` et `recette-releases` est automatique. Lors d'un commit ou d'un merge dans la branche `develop` ou `release`, TeamCity intercepte l'événement et lance automatiquement les actions ci-dessous :

- Compilation du code
- Exécution des tests unitaires
- Création des fichiers de publication
- Création du package de publication
- Upload de la mise à jour sur Lambda
- Création d'une version et déploiement sur l'environnement correspondant

Ces actions s'enchaînent sauf en cas d'erreur dans l'une d'elle. Dans ce cas, la build est en échec et les actions s'interrompent.

Pour la branche `master` et l'environnement de production, le processus de déploiement est lancé manuellement par l'équipe DevOps.

Pendant mon alternance, je n'ai pas eu accès au portail TeamCity, parce qu'il était géré et utilisé par l'équipe DevOps. En plus, je travaillais principalement sur un tout nouveau projet dont le déploiement n'était pas encore intégré dans TeamCity. Je ne l'ai donc pas vu de l'intérieur. Cependant, j'ai vu les résultats de son fonctionnement lorsque j'ai travaillé parfois sur d'autres projets (ex. Portail, API, Gestion de Parc, Cores) qui ont été déployés par TeamCity. Lorsque je travaillais sur des bogues ou des fonctionnalités et que ma branche était fusionnée dans la branche `release` ou `develop`, je voyais le message dans Slack (qui est la plateforme de communication utilisée par les équipes R&D) que le déploiement avait commencé et plus tard un autre message sur le succès ou l'échec du déploiement. Après un déploiement réussi, bien sûr, j'ai vu aussi que les modifications que j'ai faites ont été appliquées dans l'environnement `recette-releases` ou `recette-develop`.

4.4.1 Le déploiement des paquets de projet Cores

Le déploiement des paquets PHP dans le projet Cores est bien sûr différent du déploiement des autres projets. La procédure est la suivante :

1. **Le développeur met à jour le paquet dans GitLab** : Un développeur apporte des modifications et des mises à jour au code du paquet, éventuellement en corrigeant des bogues ou en ajoutant de nouvelles fonctionnalités. Il enregistre ces modifications dans le dépôt GitLab.
2. **Augmentation de la version du paquet Composer** : Dans le fichier `composer.json` du paquet, le développeur incrémente le numéro de version du paquet. Il s'agit d'une étape importante pour que Composer reconnaisse qu'une mise à jour a eu lieu.
3. **Notifier Satis** : Déclencher le processus de mise à jour pour Satis en exécutant le script créé à cet effet.
4. **Processus de mise à jour de Satis** : Le déclenchement du processus de mise à jour de Satis implique l'exécution d'une commande qui régénère les métadonnées statiques de Composer pour le dépôt Satis. Cette commande récupère les dernières informations du dépôt GitLab et met à jour les métadonnées en conséquence.
5. **Paquet mis à jour dans le dépôt Satis** : Le paquet mis à jour avec sa nouvelle version est maintenant inclus dans le dépôt Satis. Cela signifie que le paquet mis à jour est maintenant disponible pour l'installation via Composer à partir du dépôt privé Satis.
6. **Composer Update** : Les développeurs travaillant sur des projets qui dépendent de ce paquet mis à jour peuvent maintenant utiliser Composer pour mettre à jour leurs dépendances. Lorsqu'ils lancent `composer update` ou `composer require`, Composer vérifie le dépôt Satis, trouve la nouvelle version du paquetage mis à jour, le télécharge et l'installe dans leur projet.

Comme nous pouvons le constater, ce processus n'est pas encore automatisé à l'exception du fichier script pour notifier Satis. Dans ce processus manuel, il est de la responsabilité du développeur ou de l'équipe DevOps de notifier le Satis et de déclencher la mise à jour.

4.4.2 Le déploiement du projet Pipeline documentaire

Comme je l'ai mentionné précédemment, étant un nouveau projet, le déploiement du projet Pipeline documentaire n'est pas encore intégré dans TeamCity, il est fait de manière semi-automatique par l'équipe DevOps. Il peut être qualifié de semi-automatique car le projet fonctionne dans des conteneurs Docker² pendant le

2. Docker est une plateforme de virtualisation légère et portable qui permet de créer, déployer et exécuter des applications dans des conteneurs. Les conteneurs sont des environnements isolés qui regroupent tous les éléments nécessaires à l'exécution d'une application, tels que le code, les bibliothèques et les dépendances. Cela garantit une cohérence entre les environnements de développement, de test et de production, simplifiant ainsi le processus de déploiement.

développement ainsi qu'en production, il a donc un `Dockerfile`,³ un fichier `docker-compose.yaml`⁴ et l'équipe DevOps a créé un fichier de script d'installation aussi. Je décrirai brièvement ces fichiers et à travers eux le processus de déploiement.

Le fichier de script se présente comme suit (Code source 4.1).

Code source 4.1 – Le script bash pour installer le projet de Pipeline documentaire en production.

```

1  #!/bin/bash
2
3  DOCKER_VOLUME_PATH="/var/lib/docker/volumes/"
4  APP_PATH="/var/www/apps/pipeline-doc/pipelinedoc-docker"
5  DATE=$( date '+%F_%H-%M-%S' )
6  VOLUME_NAME="pipelinedoc-docker_pipeline-doc"
7  CODE_PATH="/apps/pipelinedoc-source"
8
9  # Rename volume
10 mv "${DOCKER_VOLUME_PATH}${VOLUME_NAME}" \
11   "${DOCKER_VOLUME_PATH}${VOLUME_NAME}.rollback.${DATE}"
12
13 # Update app code
14 cd "${APP_PATH}${CODE_PATH}"
15 git pull
16
17 # Restart containers
18 cd "${APP_PATH}/"
19 docker-compose down
20 docker volume rm -f "${VOLUME_NAME}"
21 docker-compose up -d --build

```

Comme on peut le voir, ce code est un script Bash conçu pour faciliter la gestion de l'application utilisant Docker. Les principales étapes du script sont les suivantes :

1. **Préparation des variables et chemins** : Les variables telles que `DOCKER_VOLUME_PATH`, `APP_PATH`, `VOLUME_NAME`, et `CODE_PATH` sont utilisées pour stocker les chemins vers les emplacements pertinents sur le système de fichiers. Par exemple, `DOCKER_VOLUME_PATH` indique le répertoire des volumes Docker et `APP_PATH` pointe vers le répertoire de l'application.
2. **Sauvegarde du volume actuel** : Avant toute modification, le script renomme le volume Docker en y ajoutant la marque "rollback" et la date actuelle. Cela crée une sauvegarde du volume actuel, ce qui peut s'avérer utile en cas de problèmes.
3. **Mise à jour du code source** : Le script se déplace dans le répertoire du code source de l'application et effectue une opération `git pull` pour mettre à jour
3. Un `Dockerfile` est un fichier texte qui contient les instructions pour construire une image Docker. Une image Docker est un modèle d'environnement qui contient tous les composants nécessaires pour exécuter une application. Le `Dockerfile` décrit les étapes pour installer les dépendances, copier les fichiers de l'application et configurer l'environnement. Une fois le `Dockerfile` créé, il peut être utilisé pour générer une image Docker prête à être exécutée dans un conteneur.
4. Le fichier `docker-compose.yaml` est un autre élément clé dans l'écosystème Docker. Il permet de définir et de gérer plusieurs conteneurs interconnectés en tant qu'application unique. Ce fichier décrit les services, les images, les ports exposés et les liens entre les conteneurs, facilitant ainsi le déploiement et la gestion d'applications complexes à plusieurs composants.

le code à partir le dépôt GitLab. Cela permet de s'assurer que l'application utilise la dernière version du code source.

4. **Arrêt des conteneurs existants** : Les conteneurs définis dans le fichier de composition Docker sont arrêtés à l'aide de la commande `docker-compose down`. Cela permet de libérer les ressources et de préparer l'environnement pour les mises à jour et les changements.
5. **Suppression du volume précédent** : Le volume Docker qui avait été renommé en début de script est supprimé à l'aide de `docker volume rm`. Cela permet de nettoyer l'espace en supprimant le volume précédent.
6. **Construction et lancement des conteneurs** : Enfin, le script utilise `docker-compose up` pour construire et lancer les conteneurs Docker, en utilisant les instructions du fichier de composition Docker. Les conteneurs sont exécutés en arrière-plan (`-d`) et les images sont reconstruites au besoin (`--build`).

En résumé, le script facilite le processus de mise à jour de l'application en effectuant des étapes clés telles que la sauvegarde des volumes, la mise à jour du code source, l'arrêt et la suppression des conteneurs, puis la construction et le lancement de nouveaux conteneurs avec la version mise à jour de l'application.

Le fichier suivant utilisé dans le processus de déploiement est le fichier `docker-compose.yaml` (Code source 4.2).

Code source 4.2 – Le fichier `docker-compose.yaml` du projet Pipeline documentaire utilisé en production.

```

1  version: "3.6"
2  services:
3    pipelinedoc:
4      build:
5        context: .
6        dockerfile: app.dockerfile
7        container_name: pipelinedoc
8        volumes:
9          - pipeline-doc:/var/www/apps/
10         - .env:/var/www/apps/pipelinedoc-source/.env
11         - ./queue-listen.conf:/etc/supervisor/conf.d/queue-listen.conf
12         - ./php-fpm.conf:/etc/supervisor/conf.d/php-fpm.conf
13         - ./supervisord.conf:/etc/supervisord.conf
14      networks:
15        - app-network
16      ports:
17        - "9001:9001"
18      deploy:
19        resources:
20          limits:
21            cpus: '0.3'
22            memory: 800M
23          reservations:
24            cpus: '0.0005'
25            memory: 100M
26
27    webserver:
28      image: nginx:stable-alpine
29      container_name: webserver
30      ports:
```

```

31      - "80:80"
32      - "443:443"
33
34 networks:
35   - app-network
36
37 volumes:
38   - ./nginx/conf.d/:/etc/nginx/conf.d/
39   - pipeline-doc:/var/www/apps/
40   - /etc/letsencrypt/live/corp.suivideflotte.net/:/etc/nginx/certs
41 depends_on:
42   - pipelinedoc
43   - database
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65

```

Ce fichier `docker-compose.yaml` définit la configuration de déploiement de plusieurs services qui font partie de l'application Pipeline documentaire en utilisant Docker Compose.

Le fichier utilise la version 3.6 de Docker Compose et configure trois services principaux : `pipelinedoc`, `webserver` et `database`. Chaque service est configuré avec ses propres caractéristiques.

1. Service `pipelinedoc` :

- Construit une image à partir d'un Dockerfile `app.dockerfile`.
- Utilise le nom de conteneur `pipelinedoc`.
- Montre des volumes pour partager des données avec le conteneur :
 - `pipeline-doc` : Volume partagé avec le chemin `/var/www/apps/` dans le conteneur.
 - `.env` : Fichier de variables d'environnement Laravel partagé à l'emplacement `/var/www/apps/pipelinedoc-source/.env`.
 - Divers fichiers de configuration pour Supervisor⁵ et Supervisord.

⁵. Supervisor est un système de gestion de processus pour les systèmes Unix-like. Il permet de

- Utilise un réseau nommé `app-network`.
 - Expose le port 9001 de Supervisor.
 - Définit des limites de ressources pour le déploiement.
2. **Service webserver :**
- Utilise l'image Nginx Alpine.
 - Utilise le nom de conteneur `webserver`.
 - Expose les ports 80 et 443. L'application Pipeline documentaire est disponible sur ces ports. Actuellement, les requêtes arrivant au port 80 (`http`) sont redirigées vers le port 443 (`https`), ce qui rend la communication plus sécurisée. Ceci est configuré dans le fichier de configuration de Nginx.
 - Montre des volumes pour partager des données avec le conteneur :
 - Configuration Nginx à partir de `./nginx/conf.d/`. Le fichier est disponible dans les Annexes (Code source A.1, page 63).
 - `pipeline-doc` : Volume partagé avec le chemin `/var/www/apps/` dans le conteneur.
 - Certificats SSL depuis `/etc/letsencrypt/live/corp.suivideflot_te.net/`.
 - Dépend des services `pipelinedoc` et `database`.
 - Utilise le réseau `app-network`.
3. **Service database :**
- Utilise l'image MariaDB version 10.
 - Utilise le nom de conteneur `database`.
 - Montre un volume nommé `dbdata` pour stocker les données MySQL.
 - Expose le port 3306.
 - Configure des variables d'environnement pour le service MariaDB.⁶
 - Utilise le réseau `app-network`.

En outre, ce fichier configure le réseau `app-network` en tant que réseau de pontage et définit deux volumes nommés `dbdata` et `pipeline-doc`, utilisant respectivement le pilote de volume local. Ce fichier `docker-compose.yaml` permet de gérer l'infrastructure nécessaire pour exécuter l'application Pipeline documentaire composée de ces trois services : `pipelinedoc`, `webserver` et `database`.

Le dernier fichier que je présente ici est le fichier `app.dockerfile` Dockerfile utilisé dans le service `pipelinedoc` (Code source 4.3).

Code source 4.3 – Le fichier `app.dockerfile` du projet Pipeline documentaire utilisé en production.

```

1 FROM php:8.1-fpm-alpine
2
3 WORKDIR /var/www/apps/pipelinedoc-source/
4 RUN mkdir /var/log/supervisor
5 RUN docker-php-ext-install pdo pdo_mysql
6 RUN apk add --update supervisor && rm -rf /tmp/* /var/cache/apk/*
7
8 RUN apk add --no-cache --virtual .build-deps $PHPIZE_DEPS \

```

contrôler et de surveiller les processus en arrière-plan, de gérer leur redémarrage en cas d'échec et de faciliter la gestion des tâches automatisées. Supervisor assure ainsi la stabilité et la disponibilité des applications en s'occupant de leur exécution continue.

6. Le mot de passe indiqué ici n'est bien sûr pas le vrai mot de passe.

```

9    && apk add --update linux-headers \
10   && pecl install xdebug-3.2.0 \
11   && docker-php-ext-enable xdebug \
12   && apk del -f .build-deps
13
14 RUN echo "xdebug.start_with_request=yes" >>
15   /usr/local/etc/php/conf.d/xdebug.ini \
16   && echo "xdebug.mode=debug,develop,coverage" >>
17   /usr/local/etc/php/conf.d/xdebug.ini \
18   && echo "xdebug.log=/var/www/html/xdebug/xdebug.log" >>
19   /usr/local/etc/php/conf.d/xdebug.ini \
20   && echo "xdebug.discover_client_host=1" >>
21   /usr/local/etc/php/conf.d/xdebug.ini \
22   && echo "xdebug.client_port=9000" >> /usr/local/etc/php/conf.d/xdebug.ini \
23   && echo "xdebug.max_nesting_level=512" >>
24   /usr/local/etc/php/conf.d/xdebug.ini
25
26 COPY --from=composer:2.4.4 /usr/bin/composer /usr/local/bin/composer
27
28 USER root
29 COPY ./apps/pipelinedoc-source .
30
31 ADD supervisord.conf /etc/
32
33 RUN composer install
34 RUN chown www-data:www-data -R storage/
35 RUN php artisan config:cache
36
37 EXPOSE 9000
38 EXPOSE 9001
39
40 ENTRYPOINT ["supervisord", "--nodaemon", "--configuration",
41   "/etc/supervisord.conf"]
42 RUN php artisan config:cache

```

Ce Dockerfile est destiné à construire une image Docker pour l'application utilisant PHP 8.1 avec FPM (FastCGI Process Manager) sur Alpine Linux.

1. **Base de l'image et répertoire de travail :** L'image est basée sur `php:8-jessie-fpm-alpine` et le répertoire de travail est défini comme `/var/www/apps/pipelinedoc-source/`.
2. **Installation des dépendances et configuration :**
 - Crée un répertoire `/var/log/supervisor` pour les journaux de Supervisor.
 - Installe les extensions PHP nécessaires (`pdo` et `pdo_mysql`).
 - Ajoute le gestionnaire de processus `supervisor` et nettoie les fichiers temporaires.
3. **Installation et configuration de Xdebug :**
 - Installe les dépendances nécessaires pour compiler des extensions PHP.
 - Installe Xdebug version 3.2.0 et l'active.
 - Configure les paramètres Xdebug dans le fichier `xdebug.ini`.
4. **Utilisateur et copie du code source :**
 - Change l'utilisateur courant en `root`.

- Copie le contenu du répertoire `./apps/pipelinedoc-source` (dans le répertoire local) dans le répertoire de travail de l'image.
5. **Installation des dépendances PHP :**
- Exécute `composer install` pour installer les dépendances PHP de l'application.
 - Modifie les permissions des fichiers/dossiers dans le répertoire `storage/` pour qu'ils appartiennent à l'utilisateur `www-data`.
 - Exécute `php artisan config:cache` pour mettre en cache les configurations Laravel.
6. **Exposition des ports :** Expose les ports 9000 et 9001, qui sont utilisés par PHP et Supervisor respectivement.
7. **Point d'entrée :** Définit `supervisord` comme point d'entrée avec certains arguments pour démarrer les processus gérés par Supervisor et laisser l'application en cours d'exécution.

En résumé, ce Dockerfile construit une image Docker pour l'application Pipeline documentaire basée sur Laravel. Il installe les dépendances, configure Xdebug pour le débogage, copie le code source de l'application, installe les dépendances PHP et configure les fichiers nécessaires, puis expose les ports requis et définit l'entrée pour démarrer les processus gérés par Supervisor.

5 Cahier des charges

L'année dernière, la société constatait sur ses solutions une véritable disparité de méthodologies d'intégration pour les documents. Chaque fonctionnalité intégrait les documents de manière différente, ce qui les amenait à constater qu'il était alors compliqué de répondre rapidement à ce type de besoins.

De plus, elle avait d'une part des demandes d'importation de nouveaux fournisseurs de carburant, mais également d'autre part la création du poste de CSM (Customer Success Manager). En outre, suite à un besoin exprimé par l'un de ses grands clients, il y avait de plus en plus de besoins pour intégrer rapidement et efficacement de nouvelles typologies de documents.

Basé sur ce constat, l'entreprise a donc décidé qu'il était nécessaire de construire un outil permettant de répondre à ces besoins : le Pipeline documentaire.

Son projet était qu'à terme, cet outil leur permettrait de généraliser l'intégration des fichiers dans leur solution. Il serait donc le point d'entrée central pour tous les documents entrant dans chacun de leurs logiciels. L'ajout d'une nouvelle typologie de document serait simple et se ferait, pour la partie intégration du moins, principalement à travers le paramétrage.

La section suivante montre comment le propriétaire du produit a exprimé les besoins concernant le Pipeline documentaire.

5.1 Expression des besoins

5.1.1 Concepts généraux

Dans sa conception, ce nouvel outil devra être le plus détaché possible du reste de l'ensemble applicatif, afin d'être ensuite mis à disposition et utilisé par l'ensemble de nos logiciels, de manière simple et intégrée. Dans cette optique, l'outil devra donc être conçu pour être notre première « Machine », concept abordé lors de notre étude de la refonte globale, qui est donc un outil logiciel mis à disposition de l'ensemble des autres applications, mais qui reste complètement autonome.

5.1.2 Composition de l'outil

Déclencheurs

Le concept de déclencheur devra faire partie intégrante de la conception de l'outil, en suivant toujours les mêmes règles et en permettant de déclencher une intégration. Lorsqu'un déclencheur est « appelé », il lancera l'intégration complète. Celui-ci fournira au guide toutes les informations nécessaires ainsi que le ou les fichiers à

intégrer. Cependant, chaque déclencheur sera indépendant et répondra à une intégration spécifique. Un certain nombre de déclencheurs ont été identifiés dans le schéma architectural (Figure 5.1). Cependant, cette liste n'est pas exhaustive.

Guide

Dans notre outil, le guide sera le point central de l'intégration. Son rôle consistera à charger la configuration adéquate afin de réaliser l'intégration, puis à faire passer le ou les fichiers à travers la chaîne d'intégration, en fonction de la configuration préalablement mise en œuvre. En fin de compte, le guide aura pour objectif de piloter l'intégration complète du fichier, c'est-à-dire de s'assurer de son bon parcours à travers la chaîne d'intégration.

Configurations

Les fichiers de configuration devront être créés pour déclarer tout nouvel import. Chaque fichier de configuration aura un format informatique, mais restera lisible par un développeur (json, yml, etc.). Ces fichiers de configuration permettront, pour chaque type d'import, de définir ses spécificités et ainsi de déterminer chaque étape de l'import. Les principales informations incluses dans ces fichiers seront les suivantes :

- Le mappage des données
- Les détails du fichier (encodage, extension, etc.)
- Le type (csv, xml, pdf, etc.)
- Les détails de la chaîne d'intégration (classes à utiliser, OCR à effectuer, etc.)
- Les traitements spécifiques à appliquer au fichier (classe de lecture spécifique, classe de mappage spécifique, etc.)
- ...

Chaine d'intégration

Le guide aura pour but de déterminer, en se basant sur les informations du déclencheur et sur la configuration chargée, l'ensemble des maillons constituant la chaîne d'intégration. Celle-ci sera composée d'étapes distinctes et autonomes, permettant d'aboutir à l'intégration complète du fichier. Le guide s'assurera que le ou les fichiers passent bien à travers chaque maillon de la chaîne d'intégration.

Actions transverses

Les actions transverses devront être automatisées et aucune configuration ni code ne devra être intégré dans les parties spécifiques d'une importation. Cela signifie que le Pipeline documentaire en lui-même devra les gérer par défaut, sans intervention du développeur qui ajoute une nouvelle chaîne, un nouveau déclencheur ou une nouvelle configuration.

Les actions transverses seront les suivantes :

- **Documentation automatique** : les configurations seront automatiquement analysées et généreront une documentation exploitable par un opérateur externe à la R&D, tel que le support ou le commerce.

- **Tableau de bord** : un ensemble de tableaux devra rendre compte du bon fonctionnement du pipeline en temps réel, ainsi qu'un récapitulatif des intégrations récentes.
- **Sauvegardes sérialisées** : l'ensemble des étapes devra être sauvegardé et sérialisé à des fins d'analyse. Cet historique pourra être périodiquement purgé. Cependant, il est intéressant de conserver le fichier d'entrée et sa configuration attachée plus longtemps que la sérialisation des étapes.

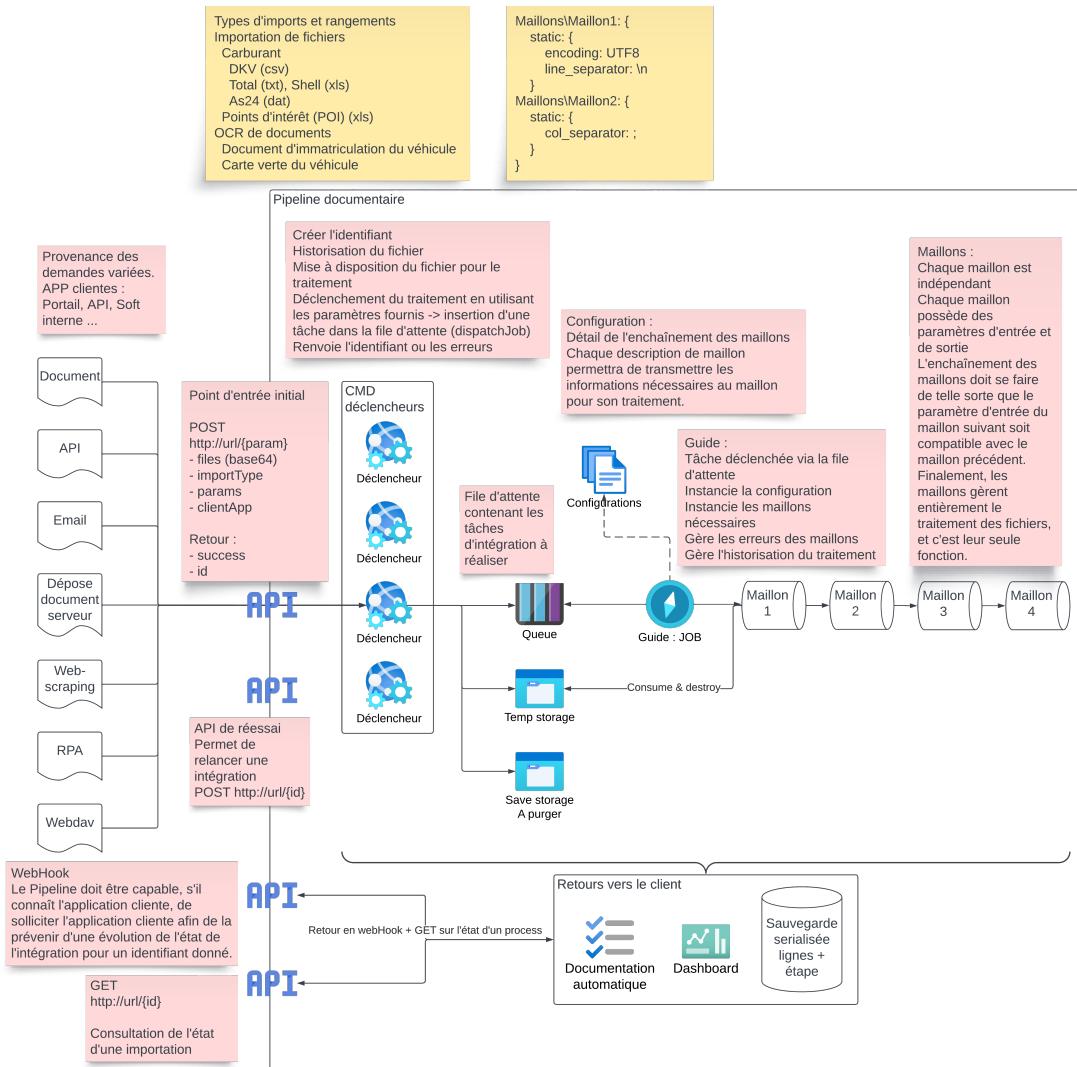


FIGURE 5.1 – Schéma architectural de Pipeline documentaire.

6 Les spécifications fonctionnelles et techniques

J'ai travaillé sur le projet Pipeline documentaire sous la direction de ma tutrice au sein de l'équipe de développement de SuiviDeFlotte. Avant de commencer, étant donné la complexité du projet, nous avons tenu plusieurs réunions avec le propriétaire du produit. Celui-ci nous a expliqué en détail ses idées concernant le projet, et bien sûr, nous avons également partagé nos réflexions.

Dans le chapitre précédent, je crois que c'était évident que notre propriétaire de produit avait une vision précise de la manière dont l'application devait être construite et de son fonctionnement. Par conséquent, nous n'avons pas eu besoin de débuter par l'analyse d'un cahier des charges pour identifier les entités et les relations qu'il pourrait contenir. Cette étape avait déjà été accomplie.

Il a également été convenu que nous développerions ce projet en utilisant le framework Laravel et que nous le déployerions dans des conteneurs Docker. Nous avons constaté que le framework Laravel offrait tous les éléments nécessaires pour ce projet : commandes, gestion des événements, mise en file d'attente des tâches, ainsi que des fonctionnalités pour créer facilement une API, entre autres. De ce fait, ce choix nous est apparu comme judicieux, d'autant plus qu'il avait déjà été utilisé pour de nombreux autres projets au sein de la société, ce qui signifiait que les développeurs possédaient une solide expérience avec cet outil.

L'utilisation de conteneurs Docker pour le développement n'était pas encore une pratique courante au sein de l'entreprise. Cependant, une transition était prévue pour que tous les projets soient conteneurisés. Par conséquent, lors du lancement de nouveaux projets, leur conteneurisation était envisagée dès le début.

Au cours de ces réunions, de plus en plus de détails ont été précisés et nous étions prêts à entamer le projet. Néanmoins, il restait encore un travail de conception à accomplir pour définir en détail l'implémentation de ce projet dans ces cadres.

Le plan de mise en œuvre du projet dans le système de contrôleurs, de commandes et d'événements de Laravel est présenté dans le diagramme d'activités de la Figure 6.1. Comme le montre le diagramme, la réalisation du projet dans Laravel nécessite un élément qui n'est présent qu'implicitement dans le schéma architectural, à savoir l'écouteur d'événements entre les déclencheurs et la file d'attente.

Le diagramme de classes permet d'obtenir une vue plus détaillée du fonctionnement du projet (Figure 6.3). Le même processus que celui illustré dans le diagramme architectural et le diagramme d'activité peut être suivi ici également, en commençant par les déclencheurs (la classe abstraite *UploadTrigger* et ses classes enfants). Au début du processus d'importation, ces classes instancient un modèle *Upload*, l'enregistrent dans la base de données et lancent l'événement *NextStepEvent*. L'écouteur d'événements (classe *StepEventSubscriber*) est à l'écoute de cet événement et ins-

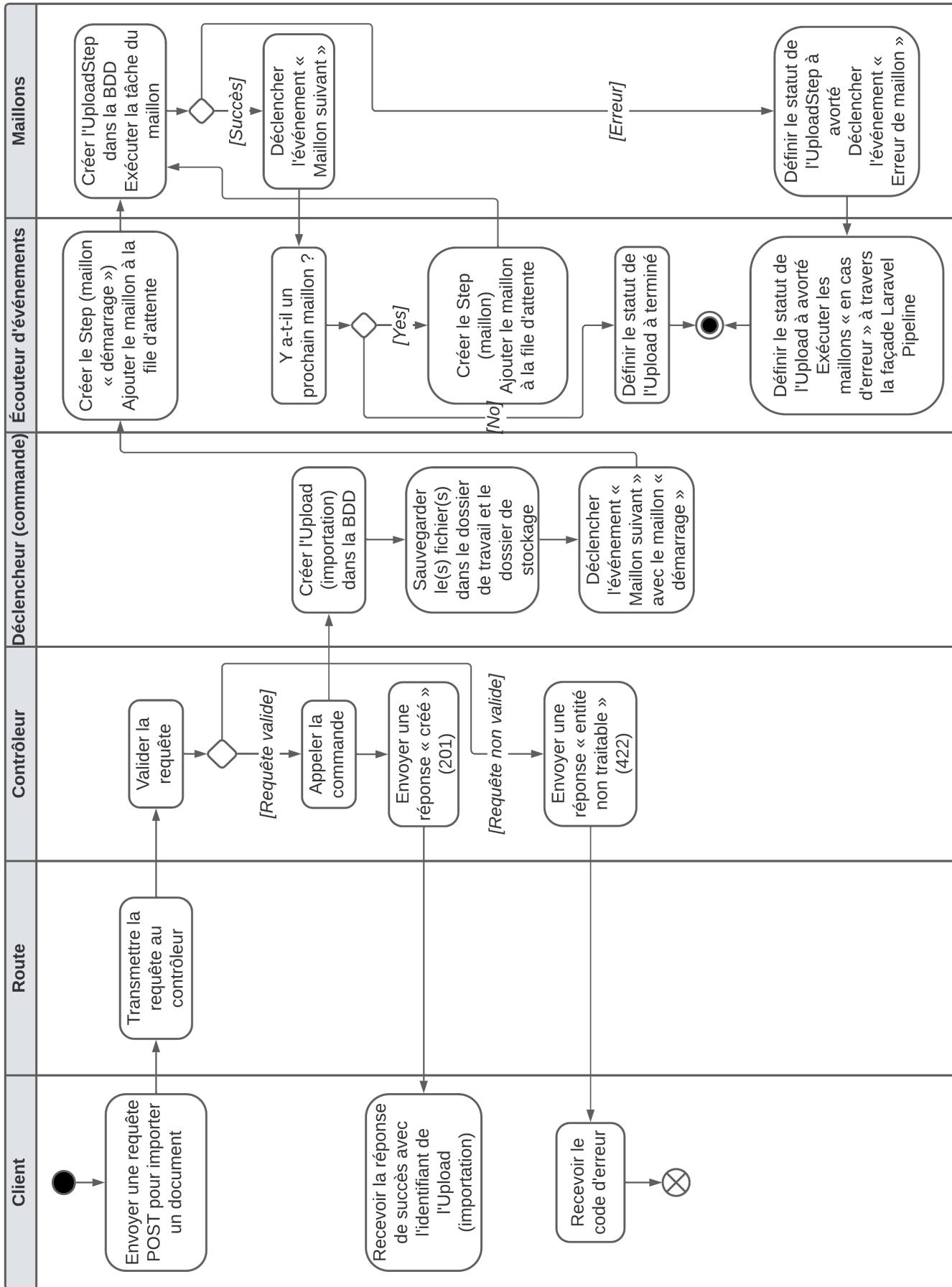


FIGURE 6.1 – Diagramme d’activité de l’envoi d’une requête POST pour importer un document.

tancie l'une des classes enfants de la classe abstraite *Step* (maillon) en fonction de la configuration du type de fichier importé qui lui est fournie par la classe *UploadType*. Le maillon instancie un modèle *UploadStep*, l'enregistre dans la base de données, exécute sa propre tâche et lance à nouveau un événement *NextStepEvent* ou, en cas d'erreur, lance l'événement *StepErrorEvent*. Ensuite, le cycle recommence avec l'écouteur d'événements, lequel consulte le fichier de configuration à l'aide de la classe *UploadType* pour trouver le maillon suivant, s'il en existe un. Dans certains cas, par exemple, lorsqu'un fichier fait l'objet d'une reconnaissance optique de caractères (OCR), les résultats sont enregistrés dans la base de données à l'aide du modèle *UploadResult*. C'est la tâche du maillon *PersistResult*. Le modèle physique des données, qui comprend les tables de base de données correspondantes aux modèles *Upload*, *UploadStep* et *UploadResult*, est illustré à la Figure 6.2.

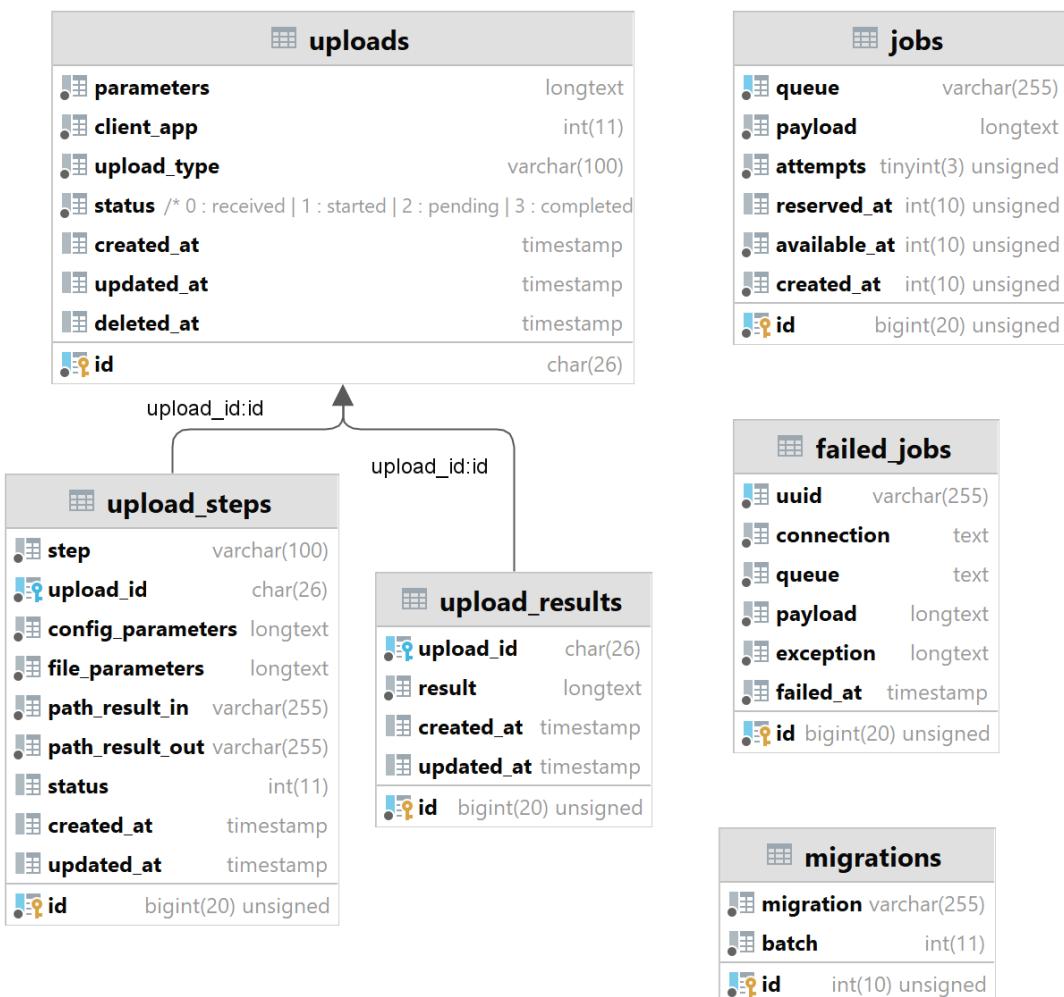


FIGURE 6.2 – Modèle physique des données du Pipeline documentaire.

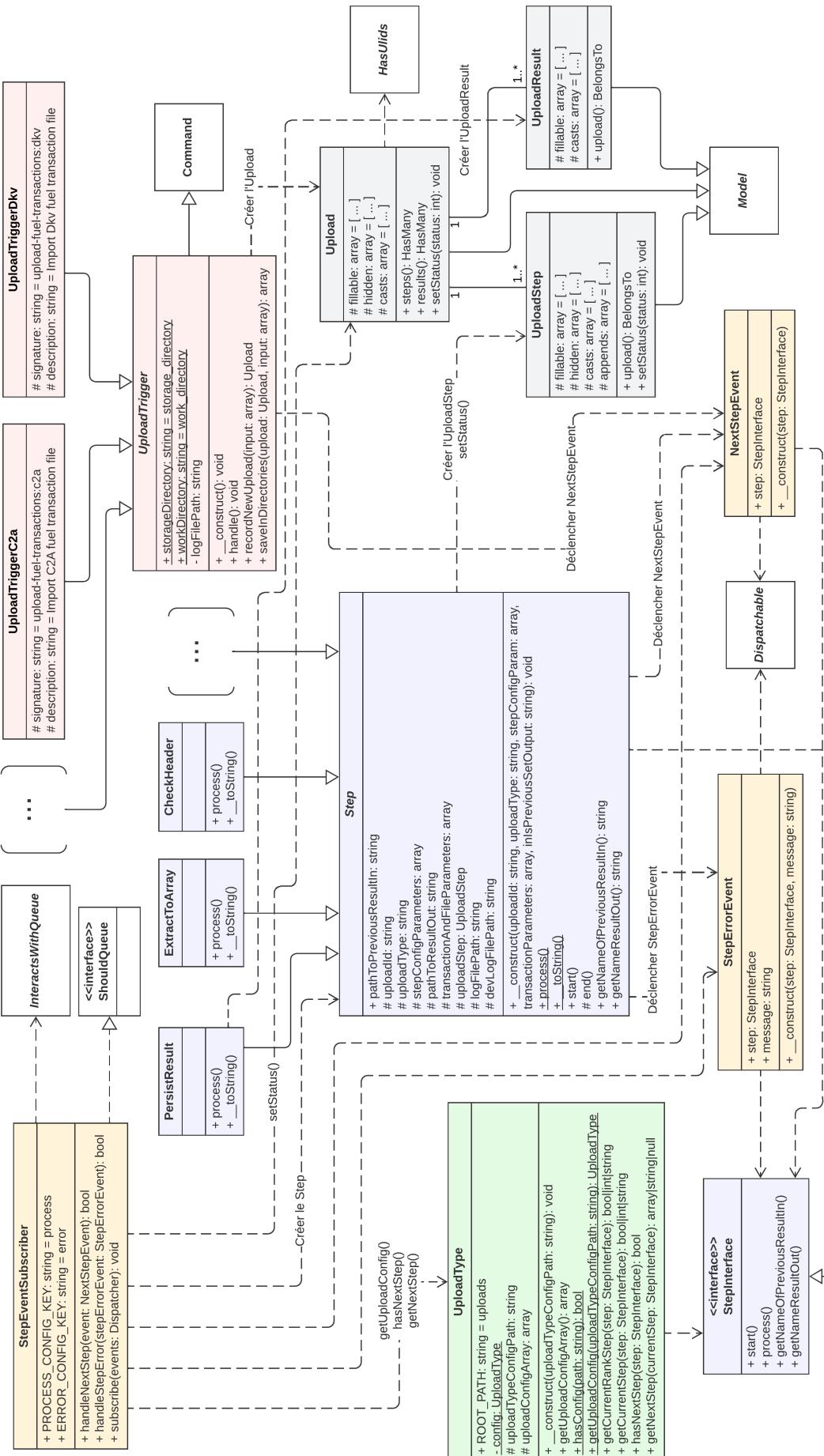


FIGURE 6.3 – Diagramme des classes principales du Pipeline documentaire.

7 Technologies

Au cours du projet Pipeline Documentaire, nous avons choisi diverses technologies pour répondre aux besoins de notre initiative (Figure 7.1). En suivant les principes agiles de la méthodologie Scrum, nous avons utilisé Jira pour documenter nos avancées et faciliter la collaboration. Nous avons rassemblé les informations essentielles liées au projet dans Confluence, ce qui a favorisé une meilleure compréhension et une communication fluide au sein de l'équipe.

Pour la gestion des versions, nous avons opté pour git et GitLab, permettant ainsi un suivi précis des modifications et une collaboration simplifiée entre les membres de l'équipe. Dans le domaine du développement, nous avons travaillé avec PHP-Storm, qui nous a fourni un ensemble d'outils adaptés à nos besoins pour la création d'applications PHP.

Afin de faciliter le déploiement, nous avons utilisé Docker et Docker Compose pour la conteneurisation du projet, ce qui a simplifié la configuration et la maintenance de l'environnement. Le cœur de notre application a été construit avec PHP 8.1 et Laravel 9, offrant une base solide pour nos développements. Nous avons utilisé MariaDB 10 comme base de données en raison de sa stabilité et de ses performances. Pour le serveur web, nous avons choisi Nginx, qui a assuré la distribution efficace de notre application et des temps de réponse rapides. Pour tester notre API, nous avons fait appel à Postman, ce qui nous a permis d'effectuer des tests approfondis pour garantir le bon fonctionnement de chaque composant.

Dans le cadre du projet Portail (Figure 4.8), j'ai travaillé sur la page d'importation des transactions de carburant. Mon objectif était de mettre à jour l'API utilisée pour cette page en utilisant la nouvelle API de Pipeline Documentaire. Pour cela, j'ai travaillé avec Vue.js et Blade dans le cadre de la mission frontend, en cherchant à offrir une expérience utilisateur améliorée et conviviale.



FIGURE 7.1 – Les principales technologies utilisées dans le cadre du projet.

8 Réalisations

Après avoir exploré en détail la conception et la structure de l'API Pipeline documentaire, je me tourne maintenant vers sa réalisation concrète. Ce chapitre mettra en lumière la mise en œuvre pratique du projet, en alignant chaque étape avec les concepts élaborés précédemment. Des exemples de code viendront illustrer la manière dont les idées abstraites se traduisent en fonctionnalités fonctionnelles.

La réalisation de ce chapitre se décompose en deux sections principales. La première section se concentrera sur la démonstration des fonctionnalités clés du Pipeline documentaire. Plutôt que d'expliquer chaque aspect du projet en détail, ce segment mettra en avant les fonctionnalités principales de l'API. En utilisant des extraits de code pertinents, je suivrai l'évolution des composants essentiels, en mettant en évidence leur rôle dans l'importation et le traitement de fichiers.

La seconde section se focalisera sur une mission frontend spécifique qui illustre l'intégration pratique de l'API nouvellement créée. Plus précisément, j'explorerais comment la page d'importation des fichiers de transactions de carburant dans le projet Gestion de parc (Figure 4.8, page 21) a été modifiée. Plutôt que d'utiliser l'ancienne API, cette page a été adaptée pour tirer parti de la puissance du nouveau Pipeline documentaire. Des exemples de code détaillés montreront comment ces modifications ont été apportées.

En réunissant la théorie et la pratique, ce chapitre offre un aperçu complet de la réalisation du projet, en montrant comment les concepts élaborés ont été traduits en un système fonctionnel.

8.1 Pipeline Documentaire

Cette première section plonge au cœur du projet Pipeline documentaire. Je commencerai par examiner le processus d'installation du projet dans un environnement de développement. Cela inclura la préparation de l'infrastructure nécessaire (conteneurs Docker) à l'exécution de l'API et la configuration de Laravel. De plus, je me pencherai sur les migrations de base de données indispensables pour assurer la cohérence des données et le bon fonctionnement de l'API. La section se poursuivra en explorant les modèles qui correspondent aux tables de la base de données.

Le cœur de cette section portera sur le processus clé de l'API : l'importation d'un fichier. Je décomposerai ce processus complexe en étapes gérables, en mettant en évidence les opérations importantes qui se produisent en arrière-plan lorsqu'un fichier est soumis à l'API. Des exemples de code accompagneront cette exploration, illustrant chaque étape du processus.

Enfin, j'aborderai l'aspect crucial des tests. La fiabilité et la robustesse de l'API sont essentielles pour assurer un fonctionnement sans faille.

En somme, cette première section plongera profondément dans l’API Pipeline documentaire, de sa mise en place à son fonctionnement concret. Chaque aspect exploré contribuera à la construction d’une base solide pour la section suivante, où j’examinerai l’intégration de l’API dans une mission frontend spécifique.

8.1.1 Installation pour le développement

Pour amorcer le développement de l’API, il est impératif de cloner deux projets à partir du dépôt GitLab de l’entreprise. Le premier de ces projets, intitulé `pipelinedoc-docker`, renferme un ensemble de fichiers essentiels. Ce dépôt inclut notamment un `Dockerfile` (`app.dockerfile`) ainsi qu’un fichier `docker-compose.yml`. Il abrite également une structure de dossiers (Figure 8.1) comprenant le fichier de configuration Nginx (`app.conf`), le répertoire destiné au code de l’API (`apps`) ainsi que d’autres dossiers de substitution nécessaires au projet.

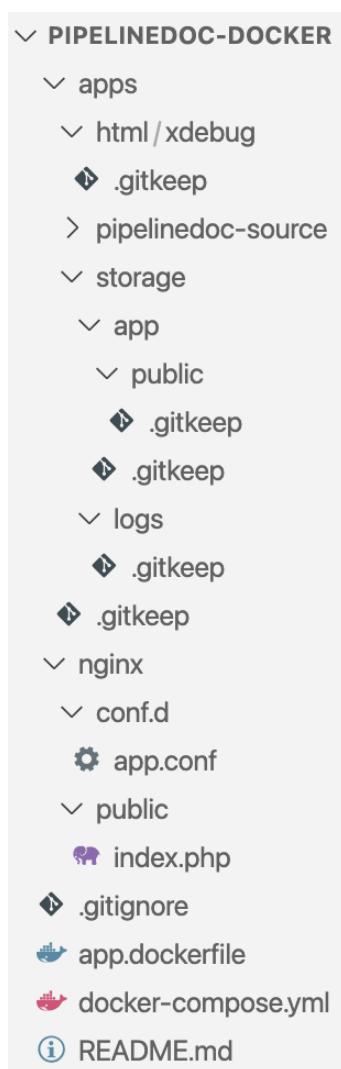


FIGURE 8.1 – La structure des dossiers du projet `pipelinedoc-docker`.

Le second projet, baptisé `pipelinedoc-source`, renferme le code source de l’API Pipeline documentaire. Ce dernier devra être cloné dans le dossier approprié au sein du projet `pipelinedoc-docker`. Cette étape est primordiale pour établir la connexion entre l’infrastructure Docker mise en place et le code de l’API à développer.

Les fichiers `app.dockerfile`, `docker-compose.yml` et le fichier de configuration Nginx sont essentiellement les mêmes que les fichiers présentés pour la production (Code source 4.3, page 28 ; Code source 4.2, page 26 ; Code source A.1, page 63). Cependant, bien qu’ils soient plus simples, ils ne contiennent pas toutes les instructions présentes dans ces fichiers.

Dès que les deux projets sont clonés, les conteneurs Docker peuvent être démarrés avec la commande suivante : `docker compose up`. Ensuite, le développeur doit créer le fichier `.env` pour le projet, contenant toutes les variables environnementales nécessaires, telles que les détails des connexions à la base de données, les URL et les jetons vers d’autres services, ainsi que d’autres paramètres de configuration. L’étape suivante consiste à installer les dépendances avec la commande `composer install` et à les mettre à jour avec la commande `composer update`. Ces commandes utilisent le fichier `composer.json` pour installer et mettre à jour les dépendances répertoriées avec leurs numéros de version. Ce projet inclut non seulement des dépendances publiques, mais également des dépendances développées et maintenues par l’entreprise dans son propre dépôt privé. Ces paquets privés sont ce que l’on appelle les « cores », comme nous l’avons mentionné précédemment (Figure 4.8, page 21 ; Sous-section 4.4.1, page 23).

Il reste encore trois petites étapes pour installer et démarrer complètement le projet : il faut lancer la commande `php artisan migrate:install`¹ pour créer la table `migrations` dans la base de données. Ensuite, il faut exécuter la commande `php artisan migrate` pour créer les tables nécessaires au projet, qui sont définies dans les fichiers de migration situés dans le dossier `database/migrations` du projet Laravel. Enfin, il faut utiliser la commande `php artisan queue:listen`² pour démarrer la file d'attente de Laravel. À ce stade, le système est prêt à importer des fichiers et à exécuter ses autres fonctionnalités. Dans les lignes qui suivent, j'examinerai un peu plus en détail les fichiers de migration qui définissent la structure des tables de la base de données nécessaires au projet.

8.1.2 Migrations de la base de données

Les migrations dans Laravel sont un mécanisme permettant de gérer et de maintenir la structure de la base de données d'une application de manière efficace et contrôlée. Les migrations définissent les schémas des tables de la base de données, ainsi que les modifications ultérieures à ces schémas. Elles offrent un moyen de collaborer entre développeurs en suivant un processus de versionnement pour les modifications de la base de données.

Pour créer une migration, on utilise l'outil de ligne de commande Artisan fourni par Laravel, en exécutant la commande `php artisan make:migration`. Cette commande génère un fichier de migration dans le répertoire `database/migrations` du projet. Dans ce fichier, on peut définir les colonnes de la table, les clés étrangères, les index, etc.

Une fois la migration créée, on peut exécuter la commande `php artisan migrate` pour appliquer les migrations en attente et mettre à jour la base de données selon les schémas définis. Les migrations permettent également de revenir en arrière en cas de besoin avec la commande `php artisan migrate:rollback`.

L'utilisation des migrations permet aux développeurs de travailler de manière collaborative et organisée en maintenant un historique des changements de structure de la base de données. Cela facilite également le déploiement sur différents environnements et assure une gestion centralisée de la structure de la base de données au sein du code source du projet.

Comme le modèle physique de données (Figure 6.2, page 35) l'a défini, nous avons créé trois tables dans la base de données à l'aide de fichiers de migration : la table des téléchargements (`uploads`), la table des étapes des téléchargements (`upload_step`) et la table des résultats des téléchargements `upload_results`. À titre d'exemple, le fichier de migration de la table `uploads` est présenté dans le Code source 8.1.

Cette migration crée une table nommée `uploads` avec plusieurs colonnes :

- `id` : Une clé primaire unique générée à l'aide d'ULID (Universally Unique Lexicographically Sortable Identifier).
- `parameters` : Une colonne de type JSON pour stocker des paramètres au format JSON.

1. Artisan est l'outil de ligne de commande intégré de Laravel, permettant d'effectuer diverses tâches liées au développement et à la gestion de projets Laravel.

2. Ces commandes doivent bien sûr être exécutées dans le conteneur de l'application et à partir de son dossier racine.

- `client_app` : Une colonne de type entier pour enregistrer l'application cliente.
- `upload_type` : Une colonne de type chaîne de caractères (string) pour spécifier le type de téléchargement.
- `status` : Une colonne de type entier avec une valeur par défaut de 0, représentant l'état du téléchargement (0 : reçu, 1 : démarré, 2 : en attente, 3 : terminé, 4 : abandonné).
- `timestamps` : Deux colonnes pour enregistrer les horodatages de création et de mise à jour des enregistrements.
- `deleted_at` : Une colonne pour prendre en charge la suppression douce (soft delete) avec une précision temporelle de 0.

Code source 8.1 – La classe de migration de la table `uploads`.

```

1 <?php
2
3 use Illuminate\Database\Migrations\Migration;
4 use Illuminate\Database\Schema\Blueprint;
5 use Illuminate\Support\Facades\Schema;
6
7 return new class extends Migration
8 {
9     /**
10      * Run the migrations.
11      *
12      * @return void
13      */
14     public function up()
15     {
16         Schema::create('uploads', function (Blueprint $table) {
17             $table->ulid('id')->primary();
18             $table->json('parameters');
19             $table->integer('client_app');
20             $table->string('upload_type', 100);
21             $table->integer('status')->default(0)->comment('0 : received | 1 :
22             → started | 2 : pending | 3 : completed | 4 : aborted');
23             $table->timestamps();
24             $table->softDeletesTz($column = 'deleted_at', $precision = 0);
25         });
26     }
27
28     /**
29      * Reverse the migrations.
30      *
31      * @return void
32      */
33     public function down()
34     {
35         Schema::dropIfExists('uploads');
36     }
};
```

La méthode `up()` est utilisée pour créer la table avec les colonnes spécifiées, tandis que la méthode `down()` est utilisée pour supprimer la table en cas de rollback. Le code est encapsulé dans une classe anonyme héritant de la classe de migration. Cela

permet de créer et exécuter la migration sans avoir à nommer explicitement la classe de migration, ce qui est courant dans les fichiers de migration Laravel.

L'objet `$table`, au sein de la classe de migration, représente l'entité qui permet de définir et de structurer la table de base de données au moyen de la migration. Dans le contexte de Laravel, cet objet est une instance de la classe `Blueprint`, qui offre un ensemble de méthodes et de fonctionnalités pour concevoir la structure de la table de manière programmatique.

Chaque méthode appelée sur l'objet `$table` correspond à une opération spécifique permettant de construire la table. Ceci facilite la création cohérente et reproductible de la base de données. Par exemple, dans ce cas, les instructions telles que `$table->ulid('id')->primary()` définissent la colonne `id` comme une clé primaire dotée d'une valeur unique générée par le biais d'ULID. De la même manière, d'autres méthodes comme `$table->json('parameters')`, `$table->integer('client_app')`, etc., définissent les colonnes de la table et leurs types de données respectifs.

L'objet `$table` revêt une importance fondamentale au sein du processus de migration Laravel. Il agit comme un canal permettant de décrire, de manière systématique, la structure de la table de base de données. Ce mécanisme offre une approche structurée et organisée pour gérer les changements de schéma de la base de données de manière réversible et documentée.

La structure des fichiers de migration des deux autres tables est similaire à celle-ci, avec des colonnes différentes bien sûr. Ces deux tables sont dans une relation un-à-plusieurs avec la table `uploads`, donc elles ont des clés étrangères comme l'une de leurs colonnes créées par le code suivant dans leurs fichiers de migration : `$table->foreignUlid('upload_id')->constrained('uploads');`

Laravel utilise les informations dans des fichiers des migrations pour générer automatiquement les requêtes SQL nécessaires. Cela se traduit par la création, la modification ou la suppression des tables et des colonnes dans la base de données sous-jacente. Par exemple, le code SQL généré par Laravel à partir du fichier de migration de la table `uploads` est présenté dans le Code source 8.2.

Code source 8.2 – Le code SQL généré par Laravel sur la base du fichier de migration de la table `uploads`.

```
1  create table `uploads` (`id` char(26) not null primary key, `parameters` json
  ↵  not null, `client_app` int not null, `upload_type` varchar(100) not null,
  ↵  `status` int default 0 comment '0 : received | 1 : started | 2 : pending |
  ↵  3 : completed | 4 : aborted', `created_at` timestamp null, `updated_at` timestamp null,
  ↵  `deleted_at` timestamp null);
```

8.1.3 Modèles

Les modèles dans Laravel sont des représentations essentielles des tables de la base de données au sein de l'application. Ils agissent comme des intermédiaires entre les données stockées dans la base de données et la logique métier de l'application. Chaque modèle est associé à une table spécifique dans la base de données et permet d'effectuer des opérations courantes, telles que la création, la lecture, la mise à jour et la suppression (CRUD), sur les données de cette table. Les modèles fournissent

également des mécanismes pour définir des relations entre les tables, comme les relations un-à-un, un-à-plusieurs et plusieurs-à-plusieurs. Grâce aux modèles, les développeurs peuvent interagir avec la base de données de manière conviviale et orientée objet, sans avoir à écrire directement des requêtes SQL complexes. En encapsulant la logique de manipulation des données, les modèles contribuent à la clarté, à la maintenabilité et à la robustesse du code de l'application Laravel.

Les modèles sont créés dans Laravel en utilisant l'outil de ligne de commande Artisan. Pour générer un modèle, on exécute la commande `php artisan make:model NomDuModèle`. Cette commande génère automatiquement une classe modèle dans le répertoire `app/Models` du projet. Cette classe étend la classe de base `Illuminate\Database\Eloquent\Model`, ce qui permet au modèle d'hériter de fonctionnalités essentielles offertes par Eloquent, le moteur de requête ORM de Laravel. Une fois le modèle créé, on peut définir les propriétés, les relations avec d'autres modèles et les méthodes spécifiques nécessaires pour interagir avec la base de données et manipuler les données associées à la table correspondante.

Comme démontré dans la sous-section précédente, nous avons créé trois tables dans la base de données à l'aide des migrations. Par conséquent, nous avons ensuite mis en place trois modèles correspondant à ces tables : `Upload`, `UploadStep` et `UploadResult`. À titre d'exemple, le code source du modèle `Upload` sera détaillé ici (Code source 8.3).

Code source 8.3 – Une version simplifiée du modèle `Upload`.

```

1 <?php
2
3 namespace App\Models;
4
5 use App\Console\Commands\UploadTrigger;
6 use App\Models\Utils\UploadStatus;
7 use Illuminate\Database\Eloquent\Cast\Attribute;
8 use Illuminate\Database\Eloquent\Concerns\HasUlids;
9 use Illuminate\Database\Eloquent\Model;
10 use Illuminate\Database\Eloquent\Relations\HasMany;
11
12 class Upload extends Model
13 {
14     use HasUlids;
15
16     protected $fillable = [
17         'parameters',
18         'client_app',
19         'upload_type'
20     ];
21
22     protected $hidden = ['client_app', 'deleted_at'];
23
24     protected $casts = ['parameters' => 'array'];
25
26     /**
27      * @return HasMany
28     */
29     public function steps(): HasMany
30     {
31         return $this->hasMany(UploadStep::class);
32     }
33 }
```

```

32 }
33
34 /**
35 * @return HasMany
36 */
37 public function results(): HasMany
38 {
39     return $this->hasMany(UploadResult::class);
40 }
41
42 /**
43 * Status of the documents upload
44 *
45 * @param int $status
46 */
47 public function setStatus(int $status)
48 {
49     $this->attributes['status'] = UploadStatus::STATUS_CODE[$status];
50     $this->save();
51 }
52
53 }

```

La classe `Upload` est située dans l'espace de noms `App\Models`. Elle utilise la fonctionnalité `HasUlids` pour gérer les ULIDs (Universally Unique Lexicographically Sortable Identifier) pour l'identifiant primaire de la table. Les propriétés `$fillable` définissent les champs de la table qui peuvent être mass-assignés et `$hidden` détermine les champs qui ne seront pas inclus lors de la conversion en tableau ou en JSON.

Le modèle utilise la fonction de casting `$casts` pour traiter le champ `parameters` comme un tableau JSON. De plus, il définit deux relations `HasMany` : `steps()` pour représenter une relation un-à-plusieurs avec le modèle `UploadStep` et `results()` pour une relation similaire avec le modèle `UploadResult`. Ces relations permettent d'accéder facilement aux étapes et aux résultats associés à un objet `Upload`.

En outre, le modèle possède une méthode `setStatus()` qui permet de définir l'état de téléchargement des documents. Cette méthode utilise la classe `UploadStatus` pour traduire le code d'état en une valeur compréhensible et met à jour la base de données avec la nouvelle valeur.

En somme, le modèle `Upload` facilite la manipulation des données de la table associée, en fournissant des méthodes pour gérer les relations et les opérations spécifiques, tout en maintenant une cohérence entre la logique de l'application et les données stockées.

8.1.4 Le processus d'importation d'un fichier

Dans cette sous-section, nous suivrons le destin d'un fichier pendant son importation, depuis son arrivée à l'API dans le corps d'une requête POST, jusqu'à l'enregistrement de ses données dans la table appropriée de la base de données adéquate. Nous utiliserons à titre d'exemple un fichier LCCC (La Compagnie des Cartes Carburant SAS) qui est un type de fichier de transactions de carburant au format `csv` (Figure 8.2).

```

1 Enseigne;Site : Code site;Site : Numéro de terminal;Site : Libellé;
Site : Libellé court;Site : Type du site;Client : Référence client;
Client : Nom du client;Carte : Type de carte;Carte : Numéro de carte;
Carte : Date de validité;Numéro de TLC;Date de télécollecte;Type de
transaction;Numéro de transaction;Date de transaction;Code devise;
Code produit;Produit;Prix unitaire;Quantité;Montant;Montant HT;Code
véhicule;Code chauffeur;Kilométrage;Immatriculation;Code réponse;
Numéro d'opposition;Numéro d'autorisation;Motif d'autorisation;Mode
de transaction;Mode de vente;Mode de validation;Facturation client;
Facturation site
2 LA COMPAGNIE DES CARTES CARBURANT SAS;'537307;'54;
07651-BARBEZIEUX-Intermarche;SITE42665;DoDo;'144127;RWT ENERGY 33;
Carburant PRO;'7824638575200010319;04-24;'856;10-02-2023;Transaction
de débit;'99996;10-02-2023 14:10:13;978;'01;B7 GAZOLE;2,109;43,50;91,
74,76,45;;'2674;'106054;;Acceptée;'0;'166945;Paramétrage Site – Appel
Systématique;Nominal On-Line Srv primaire;DAC;Lecture piste avec code;
Oui;Oui
3 LA COMPAGNIE DES CARTES CARBURANT SAS;'537307;'54;
07651-BARBEZIEUX-Intermarche;SITE42665;DoDo;'144127;RWT ENERGY 33;
Carburant PRO;'7824638575200010319;04-24;'856;10-02-2023;Transaction
de débit;'99996;11-02-2023 14:10:13;978;'11;E5 SP98;2,109;43,50;91,
74,76,45;;'2674;'106054;;Acceptée;'0;'166945;Paramétrage Site – Appel
Systématique;Nominal On-Line Srv primaire;DAC;Lecture piste avec code;
Oui;Oui

```

FIGURE 8.2 – Exemple d'un fichier LCCC.

Pour l'API, il est possible d'envoyer des fichiers dans le corps d'une requête POST au format encodé en base64, en tant que partie d'un objet JSON bien défini, comme démontré dans le Code source 8.4. Cette charge utile est validée avant d'être acceptée par l'API, plus précisément l'application Laravel utilise la méthode `rules()` de la classe `UploadStoreRequest` créée par nous pour la valider (Code source 8.6).

Cette classe se trouve dans le dossier `app/Http/Requests`. Elle étend la classe `FormRequest` de Laravel, permettant ainsi de définir les règles de validation pour la requête de stockage (store request) d'import de fichiers.

La méthode `rules()` définit les règles de validation pour les différentes parties de la requête. Les règles sont définies sous forme d'un tableau associatif. Parmi les règles définies :

- Le champ `client_app` doit être présent, être un nombre entier et être une des valeurs autorisées spécifiées dans la configuration du projet (`allowed_client_app`).
- Le champ `parameters` doit être un tableau avec au moins deux éléments.
- Les éléments spécifiques du champ `parameters` (`client_id` et `user_id`) doivent être présents, être des nombres entiers et être obligatoires.
- Le champ `files` doit être un tableau avec au moins un élément.³
- Les éléments spécifiques du champ `files` (`content`) doivent être présents,

3. Pour l'instant, l'API n'est pas en mesure de gérer plus d'un fichier. Si plusieurs fichiers sont présents dans la charge utile, ils ne sont pas pris en compte.

commencer par `data:text/plain;base64`, et avoir une longueur minimale de 27 caractères.

En somme, la classe `UploadStoreRequest` sert de mécanisme de validation pour les requêtes POST envoyées à l'API, en s'assurant que les données fournies sont conformes aux règles spécifiées, contribuant ainsi à la sécurité et à la cohérence des données entrantes.

La charge utile doit être envoyée à la route appropriée définie dans le fichier `routes/api.php` (Code source 8.5).

Code source 8.4 – La charge utile (payload) à envoyer contenant le fichier encodé au format base64 et certaines métadonnées associées.

```

1  {
2      "files": [
3          {
4              "content": "data:text/plain;base64,RW5zZWlnbmU7U2l0ZSA6IENvZGUgc2l0J
5                  ZTtTaXR1IDogTnVt6XJvIGRIIHRLcm1pbmFs01NpdGUg0iBMaWJlbGzp01NpdGU
6                  g0iBMaWJlbGzpIGNvdXJ001NpdGUg0iBUExB1IGR1IHNpdGU7Q2xpZW50IDogUu
7                  lm6XJ1bmN1IGNsaWVudDtDbG11bnQg0iB0b20gZHugY2xpZW5000NhcnR1IDogV
8                  HlwZSBkZSBjYXJ0ZTtDYXJ0ZSA6IE51belybyBkZSBjYXJ0ZTtDYXJ0ZSA6IERh
9                  dGUGZGUGdmFsaWRpd0k7TnVt6XJvIGRIIFRMQzteYXR1IGRIIHTpb01jb2xsZWN
10                 jOZTtUeXB1IGR1IHRyYW5zYWN0aW9u0051belybyBkZSB0cmFuc2FjdGlvbjtEYX
11                 R1IGR1IHRyYW5zYWN0aW9u00NvZGUGZGV2aN100NvZGUGchJvZHVPdDtQcm9kd
12                 jWl001ByaXggdW5pdGFpcmU7UXVhbnRpdk7TW9udGFudDtNb250YW50IEhU00Nv
13                 jZGUGduloaWN1bGU7Q29kZSBjaGF1ZmZ1dXI7S21sb23pdHJhZ2U7SW1tYXRyaWN
14                 j1bGF0aW9u00NvZGUGculwb25zZTt0dW3pcm8gZCdcvHBvc210aW9u0051belyby
15                 jBkJ2F1dG9yaXNhdGlvbjtNb3RpZiBkJ2F1dG9yaXNhdGlvbjtNb2R1IGR1IHRyY
16                 jW5zYWN0aW9u001vZGUGZGUGdmFsaWVudGU7TW9kZSBkZSB2YWxpZGF0aW9u00ZhY3R1
17                 jcmF0aW9uIGNsaWVudDtGYWN0dXJhdGlvb1BzaXR1DQpMQSBDT01QQUd0SUUgREV
18                 jTIENBU1RFUyBDQVJCVVJBT1QgU0FT0yc1MzcMdc7JzU00za3NjUxLUJBukJFWk
19                 j1FVVgtSW50ZXJtYXJjaGU7U01URTQyNjY100RvRG87JzEONDEyNztSV1QgRU5FU
20                 jkdZIDMz00NhcmJ1cmFudCBQuk87Jzc4MjQ2Mzg1NzUyMDAwMTAzMTk7MDQtMjQ7
21                 jJzg1NjsxMC0wMi0yMDIZ01RyYW5zYWN0aW9uIGR1IGTpYml00yc50Tk5NjsxMCO
22                 jwMi0yMDIZIDE00jEw0jEz0zk30DsnMDE7Qjcg1EdBWk9MRTsyLDEw0Ts0Myw1MD
23                 js5MSw3NDs3Niw0NTs7JzI2NzQ7JzEwNjA1NDs7QWNjZXB06WU7JzA7JzE2Njk0N
24                 jTtQYXJhb10cmFnZSBTaXR1IC0gQXBwZwggU31zd0ltYXRpcXV1005vbWluYWwg
25                 jT24tTGluzSBTcnYgcHJpbWFpcmU7REFD00x1Y3R1cmUgcG1zdGUGYZ1YyBjb2R
26                 j10091aTtPdWkNCkxBIENPTVBBR05JRSBERVMgQ0FSVEVTIENBUkJVUkFOVCBTQV
27                 jM7JzUzNzMwNzsnNTQ7MDc2NTEtQkFSQkVaSUVVWC1JbnR1cm1hcmNoZTtTSVRFN
28                 jDI2NjU7RG9EbzsmtQOMTI301JXVCBFTkVSR1kgMzM7Q2FyYnVyYW50IFBStzsn
29                 jNzgyNDYz0DU3NTiwmDAxMDMx0TswNC0yNDsn0DU20zEwLTAYLTIwMjM7VHJhbnN
30                 jhY3Rpb24gZGUGZ0liaXQ7Jzk50Tk20zExLTAYLTIwMjMgMTQ6MTA6MTM70Tc40y
31                 jcxtTfNSAgU1A50DsyLDEw0Ts0Myw1MDs5MSw3NDs3Niw0NTs7JzI2NzQ7JzEwN
32                 jjA1NDs7QWNjZXB06WU7JzA7JzE2Njk0NTtQYXJhb10cmFnZSBTaXR1IC0gQXBw
33                 jZwggU31zd0ltYXRpcXV1005vbWluYWwgT24tTGluzSBTcnYgcHJpbWFpcmU7REF
34                 jD00x1Y3R1cmUgcG1zdGUGYZ1YyBjb2R10091aTtPdWkNCg=="
35             }
36         ],
37         "client_app": 25,
38         "parameters": {
39             "client_id": 1,
40             "user_id": 12345
41         }
42     }

```

Code source 8.5 – La définition de la route d'importation de fichiers dans le fichier `routes/api.php`.

```
47 Route::post('/command/uploads/{category}/{type}', [UploadController::class,
    ↵  'upload']);
```

Cette route est définie pour une requête POST vers l'URL `/command/uploads/{category}/{type}`. Lorsqu'une requête POST est effectuée vers cette URL, elle sera traitée par la méthode `upload` du contrôleur `UploadController`. Cette méthode prendra en charge le traitement de la requête et la gestion de l'importation des fichiers correspondant à la catégorie et au type spécifiés dans l'URL. Dans le cas de notre fichier, la catégorie est `fuel-transactions`, le type est `lccc`. Cette définition de route établit une correspondance entre une URL POST spécifique et la méthode du contrôleur qui gère le traitement de la requête et les opérations nécessaires. Cela permet de créer une API RESTful qui répond aux requêtes POST adressées à cette URL particulière.

Code source 8.6 – La classe `UploadStoreRequest`.

```
1 <?php
2
3 namespace App\Http\Requests;
4
5 use Illuminate\Foundation\Http\FormRequest;
6 use Illuminate\Validation\Rule;
7
8 class UploadStoreRequest extends FormRequest
{
9
10 /**
11 * Get the validation rules that apply to the request.
12 *
13 * @return array<string, mixed>
14 */
15 public function rules(): array
16 {
17     $allowedClientApps = config('allowed_client_app');
18     return [
19         'client_app' => [
20             'required',
21             'numeric',
22             'integer',
23             Rule::in(array_keys($allowedClientApps)),
24         ],
25         'parameters' => 'required|array|min:2',
26         'parameters.client_id' => 'required|numeric|integer',
27         'parameters.user_id' => 'required|numeric|integer',
28         'files' => 'required|array|min:1',
29         'files.*.content' =>
30             'required|starts_with:data:text/plain;base64,|min:27'
31     ];
32 }
33 }
```

La méthode upload de la classe UploadController

La classe `UploadController` (Code source 8.7) représente une classe de contrôleur dans le projet. Elle gère les requêtes en répondant à certaines routes définies dans l'API. Nous ne nous occupons ici que de la route d'importation des fichiers et de la méthode `upload` de la classe. Lorsqu'une requête POST est effectuée vers cette route, la méthode `upload` du contrôleur est appelée.

Code source 8.7 – La version simplifiée de la classe `UploadController`.

```

1 <?php
2
3 namespace App\Http\Controllers;
4
5 use App\Http\Requests\UploadStoreRequest;
6 use Illuminate\Http\JsonResponse;
7 use Illuminate\Support\Facades\Artisan;
8
9 class UploadController extends Controller
10 {
11     public function upload(UploadStoreRequest $request, string $category,
12         string $type): JsonResponse
13     {
14         Artisan::call(
15             'upload-' . $category . ':' . $type,
16             [
17                 'token' => request()->header('Authorization'),
18                 'input' => $request->validated(),
19                 'category' => $category,
20                 'type' => $type
21             ]
22         );
23
24         $o = Artisan::output();
25         $o = json_decode($o, true);
26
27         return new JsonResponse($o, 201);
28     }
}

```

La méthode `upload` prend trois paramètres : une instance de la classe `UploadStoreRequest` (Code source 8.6, page 47) pour la validation de la requête, une chaîne `category` pour la catégorie du fichier et une chaîne `type` pour le type de fichier. La méthode utilise ensuite la classe `Artisan` pour appeler une commande Artisan personnalisée correspondant à l'importation du fichier en utilisant la catégorie et le type spécifiés. Les paramètres tels que le jeton d'autorisation, les données validées de la requête et les détails de la catégorie et du type sont transmis à la commande.

Après l'exécution de la commande, la sortie est récupérée et analysée en tant que JSON. Cette sortie représente le résultat de l'importation du fichier, qui est ensuite renvoyé sous forme de réponse JSON avec le code de statut 201 (Créé). En somme, la classe `UploadController` agit comme un pont entre les requêtes d'importation de fichiers, la validation des données et l'exécution de la commande associée pour traiter les fichiers importés et renvoyer les résultats appropriés.

La classe abstraite UploadTrigger et ses enfants

Dans l'API, les commandes Artisan personnalisées qui sont appelées par la méthode `upload` de la classe `UploadController` sont représentées par la classe abstraite `UploadTrigger` et ses enfants. Elles se trouvent dans le dossier `app/Console/Commands` du projet.

La classe `UploadTrigger` est une classe de commande, destinée à gérer le processus de déclenchement d'une importation de fichiers. Elle étend la classe `Command` de Laravel, fournissant ainsi la base pour définir des commandes spécifiques.

La classe contient des propriétés statiques `storageDirectory` et `workDirectory` pour les répertoires de stockage et de travail. Dans le constructeur (Code source 8.8), des arguments d'entrée (`input`, `token`, `category` et `type`) sont définis pour la commande, ainsi qu'une définition d'entrée pour les utiliser.

Code source 8.8 – Le constructeur de la classe `UploadTrigger`.

```

28     public function __construct()
29     {
30         parent::__construct();
31
32         $inputArgument = new InputArgument('input', InputArgument::REQUIRED,
33             'desc');
33         $authorizationArgument = new InputArgument('token',
34             InputArgument::REQUIRED, 'Token authorization');
34         $categoryArgument = new InputArgument('category',
35             InputArgument::REQUIRED, 'Category of the upload');
35         $typeArgument = new InputArgument('type', InputArgument::REQUIRED,
36             'Type of the upload');
36         $definition = new InputDefinition([$authorizationArgument,
37             $inputArgument, $categoryArgument, $typeArgument]);
37         $this->setDefinition($definition);
38     }

```

La méthode `handle()` (Code source 8.9) est appelée lorsque la commande est exécutée. Elle enregistre un nouvel `Upload` en utilisant les paramètres fournis, puis déclenche un événement `StepStart` avec les paramètres nécessaires pour démarrer l'étape de traitement. Le statut de l'importation est mis à jour et les informations sont enregistrées dans un fichier journal.

Code source 8.9 – La méthode `handle()` de la classe `UploadTrigger`.

```

40     public function handle(): void
41     {
42         $input = $this->argument('input');
43
44         $upload = $this->recordNewUpload($input);
45
46         $transactionAndFileParam = array_merge(
47             ['upload_type' => $this->argument('category') . '.' .
48                 $this->argument('type')],
48             $input['parameters'],
49             array_key_exists('parameters', $input['files'][0]) ?
49                 $input['files'][0]['parameters'] : [],
50                 ['token' => $this->argument('token')]
```

```

51 );
52
53     $stepParamIn = $upload->id . '_0_';
54
55     NextStepEvent::dispatch(new StepStart($upload->id,
56         $upload->upload_type, [], $transactionAndFileParam, $stepParamIn));
57
58     $upload->setStatus(UploadStatus::STARTED);
59     $uploadResource = new
60         UploadResource($upload->append('UploadStatus')->only(['id',
61             'status', 'UploadStatus']));
62
63     Logger::appendLog($this->logFilePath,
64         $uploadResource->response()->content(), LogLevel::INFO);
65     $this->info($uploadResource->response()->content());
66 }

```

La méthode `recordNewUpload()` (Code source 8.10) enregistre les détails de l'importation dans la base de données, gère les fichiers associés et renvoie un objet `Upload`.

Code source 8.10 – La méthode `recordNewUpload()` de la classe `UploadTrigger`.

```

65     public function recordNewUpload(array $input): Upload
66     {
67         $upload = Upload::create(
68             [
69                 'parameters' => $input['parameters'],
70                 'client_app' => (int)$input['client_app'],
71                 'upload_type' => $this->argument('category') . '.' .
72                     $this->argument('type')
73             ]
74         );
75
76         $this->logFilePath = $upload->id . '/' . $upload->id . '_log';
77
78         $result = $this->saveInDirectories($upload, $input);
79         $success = 0 === count(array_keys($result, false));
80         if (false === $success) {
81             $upload->setStatus(UploadStatus::ABORTED);
82             $message = 'File save failed.';
83             $code = 500;
84             $uploadResponse = new UploadResponse($upload->id, $upload->status,
85                 $message, $result, $code);
86             Logger::appendLog($this->logFilePath,
87                 $uploadResponse->getUploadJsonResponse(), LogLevel::ERROR);
88             ErrorResponseSender::sendErrorResponse($message, $code);
89         }
90         return $upload;
91     }

```

La méthode `saveInDirectories()` (Code source 8.11) gère l'enregistrement des fichiers dans les répertoires de stockage et de travail, tout en gérant les paramètres associés.

Code source 8.11 – La méthode `saveInDirectories()` de la classe `UploadTrigger`.

```

90     public function saveInDirectories(Upload $upload, array $input): array
91     {
92         $inputFiles = $input['files'];
93         $uploadParameters = $input['parameters'];
94         $result = [];
95         $payload = json_encode($input);
96
97         $basePathForUpload = $upload->id . '/';
98         $saved = Storage::disk(self::$storageDirectory)->put($upload->id . '/'
99             . $upload->id . '_payload.json', $payload);
100        $result['payload'] = $saved;
101
102        foreach ($inputFiles as $key => $file) {
103            $content = $file['content'];
104            $content = str_replace('data:text/plain;base64,', '', $content);
105            $content = base64_decode($content);
106
107            $basePath = $upload->id . '/';
108
109            $contentPath = $basePath . '/' . $upload->id . '_' . $key . '_';
110            $contentSavedInWorkDir = Storage::disk(self::$workDirectory)
111                ->put($contentPath, $content);
112
113            $paramPath = $contentPath . '_parameters';
114            $contentSavedInStorageDir =
115                Storage::disk(self::$storageDirectory)->put($contentPath,
116                    $content);
117            if (array_key_exists('parameters', $file)) {
118                $paramSavedInStorageDir =
119                    Storage::disk(self::$storageDirectory)->put($paramPath,
120                        json_encode($file['parameters']));
121            }
122            $saved = true === $contentSavedInWorkDir && true ===
123                $contentSavedInStorageDir;
124            $result['file_' . $key] = $saved;
125        }
126        return $result;
127    }

```

L’objectif global de la classe est de gérer le flux de travail lié à l’importation de fichiers, de l’enregistrement initial à l’enregistrement des fichiers et à la mise à jour des statuts et des journaux correspondants. Cette classe de commande abstraite joue un rôle essentiel dans la gestion des processus d’importation de fichiers dans l’API, en encapsulant les étapes et les opérations associées dans des méthodes clés.

Cette classe contient les fonctionnalités communes des commandes d’importation de fichiers. Pour créer des commandes spécifiques à un fichier, il convient d’étendre cette classe et de définir les attributs spécifiques dans les classes enfants.

La classe `UploadTriggerLCCC` (Code source 8.12) est une classe de commande spécifique à notre fichier d’exemple, destinée à gérer le déclenchement de l’importation des transactions de carburant spécifiques à LCCC. Elle étend la classe abstraite `UploadTrigger`, héritant ainsi des fonctionnalités et du comportement définis dans la classe parente.

La propriété `signature` est définie pour cette commande, indiquant comment la commande doit être appelée à partir de la ligne de commande Laravel ou à partir du code, comme dans la méthode `upload` de la classe `UploadController`. Dans ce cas, la signature est `upload-fuel-transactions:lccc`.

La propriété `description` donne une brève description de ce que fait la commande. Dans ce cas, la description est « Import fuel transactions LCCC », ce qui indique clairement que la commande est utilisée pour l'importation des transactions de carburant spécifiques à LCCC.

Code source 8.12 – La classe `UploadTriggerLCCC`.

```
1 <?php
2
3 namespace App\Console\Commands\FuelTransactions;
4
5 use App\Console\Commands\UploadTrigger;
6
7 class UploadTriggerLCCC extends UploadTrigger
8 {
9     protected $signature = 'upload-fuel-transactions:lccc';
10
11     protected $description = 'Import fuel transactions LCCC';
12 }
```

1. POST
 - (a) `NextStepEvent`, `StepErrorEvent`
 - (b) `StepEventSubscriber`
 - (c) `UploadType`
 - (d) Step, 1-2 exemples
2. Testing

8.2 Mission frontend

1. Maquettage
2. Modal
3. Importations

9 Présentation d'un jeu d'essai

10 Veille sur les vulnérabilités de sécurité

11 Description d'une situation de travail ayant nécessité des travaux de recherches

12 Conclusion

13 Remerciements

Annexes

A Gestión de proyecto

A.1 Jira : L'outil essentiel pour la gestion quotidienne de projet

Projets / SuivideFlotte.net / Backlog Produit

Tableau Kanban

The screenshot shows a Jira Kanban board with the following columns and tasks:

- IDÉES** (Ideas):
 - Zones de livraison d'un rayon de 10km (SDFN-381)
 - Gestion cuve de gasoil (SDFN-379)
 - Mettre les conf de RC à niveau pour SuiviMission (SDFN-12502)
 - SI : Support technique attribution de ticket (SDFN-13496)
 - Nom du groupe d'un véhicule sur la carte (SDFN-12791)
- EN RAFFINEMENT** (Refinement):
 - Ajouter acces application en mode Badge identifiant au lieu de Immatriculation (SDFN-286)
 - Alerte échéance Carte entreprise (Offre PL) (SDFN-272)
 - Documents réglementaires de conduite (SDFN-383)
 - Déclaration de sinistre via l'application mobile (SDFN-54)
 - Détection coupure moteur
- EN RAFFINEMENT TECHNIQUE** (Technical Refinement):
 - Gestion Cartes carburant (SDFN-12348)
 - Trouver l'origine des crash récurrents de php5-fpm sur le serveur de dev (2022.2)
 - Refonte de la gestion du paramètre vie privée d'un véhicule (SDFN-12151)
- PRÉTÉS** (Prepared):
 - Mettre en ligne le guide d'intégration DKV (SDFN-13565)
 - Analysé les possibilité sur les POI avec l'api Mapotempo (SDFN-13265)
 - Visualiser, dans le rapport journalier, le passage en ZFE - Partie Carte (MISE EN PLACE DE LA GESTION...)
 - Vérification de fin de charge ZOE FG-490-EA (SDFN-12822)
 - Pipelinedoc - Import dkv (PIPELINE D'IMPORT)
- EN COURS** (In Progress):
 - Refacto : Création des projets de test RPC (SDFN-13501)
 - Remplacer les appels par l'api de géocodage (REFONTE DU MÉCANISME DE G...)
 - Utilisation de l'outil de reverse géocodage dans le reste de SDF (REFONTE DU MÉCANISME DE G...)
 - Scan plaque d'immat quick privacy (PIPELINE D'IMPORT)
- EN ATTENTE** (On Hold):
 - Analysé la problématique de traitement de la queue kuantic de la semaine du 26 - (REFONTE - ALERTE DE ZONE)
 - Les filtres doivent être dynamiques et se recharger selon les selections des autres (EVOLUTIONS TCO)
 - Interactivité tableau / diagramme (EVOLUTIONS TCO)
 - Scan plaque d'immat quick privacy (PIPELINE D'IMPORT)
- TERMINÉES** (Completed):
 - Etude refonte alertes de zones (REFONTE - ALERTE DE ZONE)
 - Vérifier les utilisations de la table fournisseur legacy (Plan d'action de refonte générale back)
 - Analyse découpe BDD (REFONTE TECHNIQUE - DÉCOU...)

FIGURE A.1 – Le Backlog de produit sous forme de tableau Kanban dans Jira.

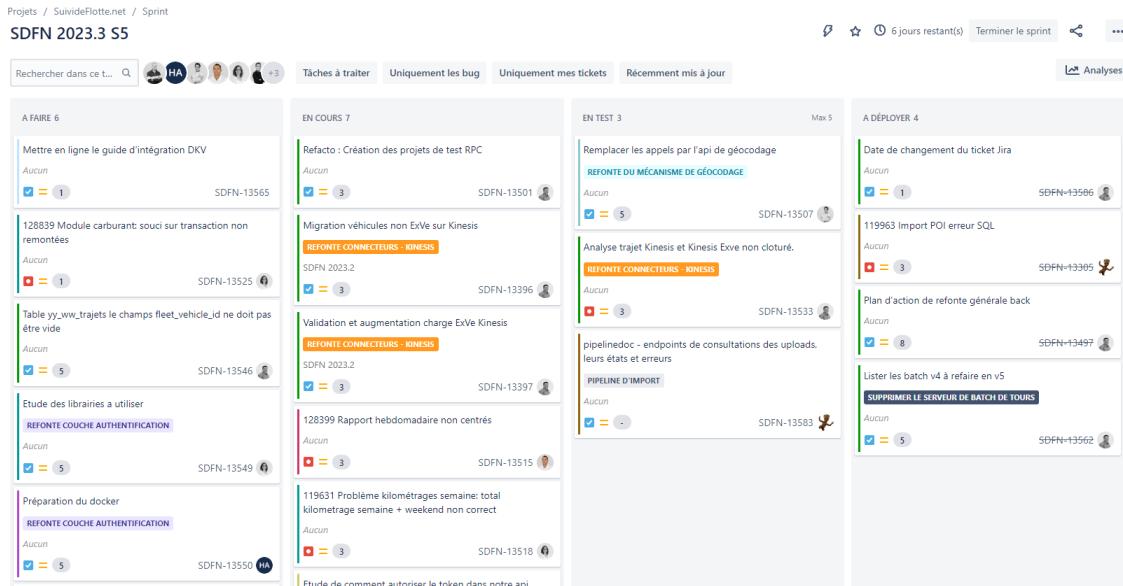


FIGURE A.2 – Le Backlog de sprint sous forme de tableau Kanban dans Jira.

The screenshot shows a detailed view of a Jira ticket. The ticket title is 'Front : Implementation du design de maquette - affichage modale'. The status is 'Terminé'. The ticket has a responsible person (adrien.bacarisse), a reporter (Csaba SCHNITCHEN), and a developer (Csaba SCHNITCHEN). The ticket is associated with a story point of 5 and an epic link to 'Refonte de l'écran de trans...'. The ticket was created on May 24, 2023, at 08:47. The ticket details include the responsible person, reporter, developer, and tester information, along with the ticket's history and activity feed.

FIGURE A.3 – Un exemple de ticket dans Jira pour le projet d'amélioration de la page d'importation des fichiers des transactions de carburant.

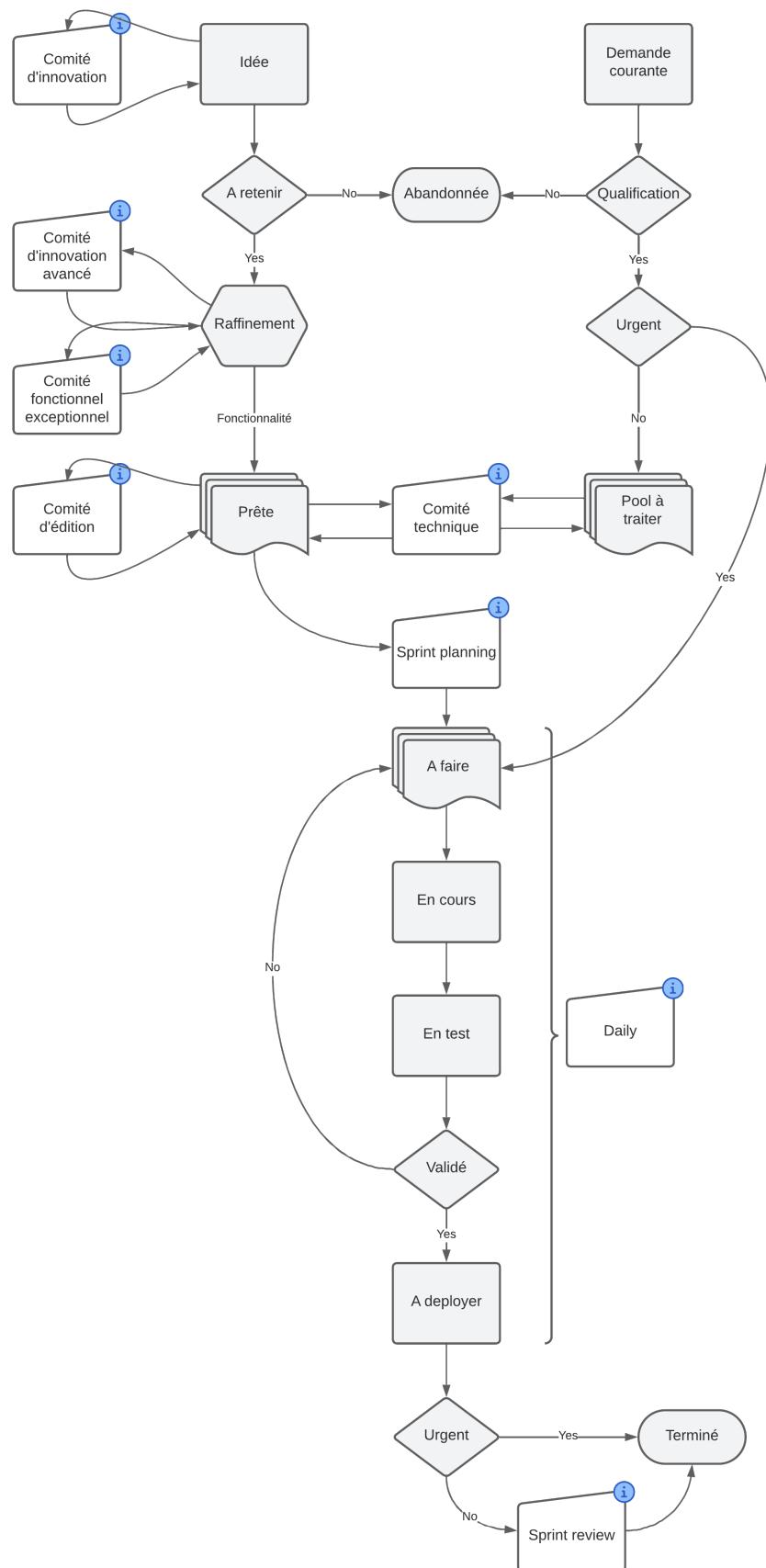


FIGURE A.4 – Diagramme d'activité du traitement des idées et des bogues (demandes courantes) entrants chez SuiviDeFlotte.

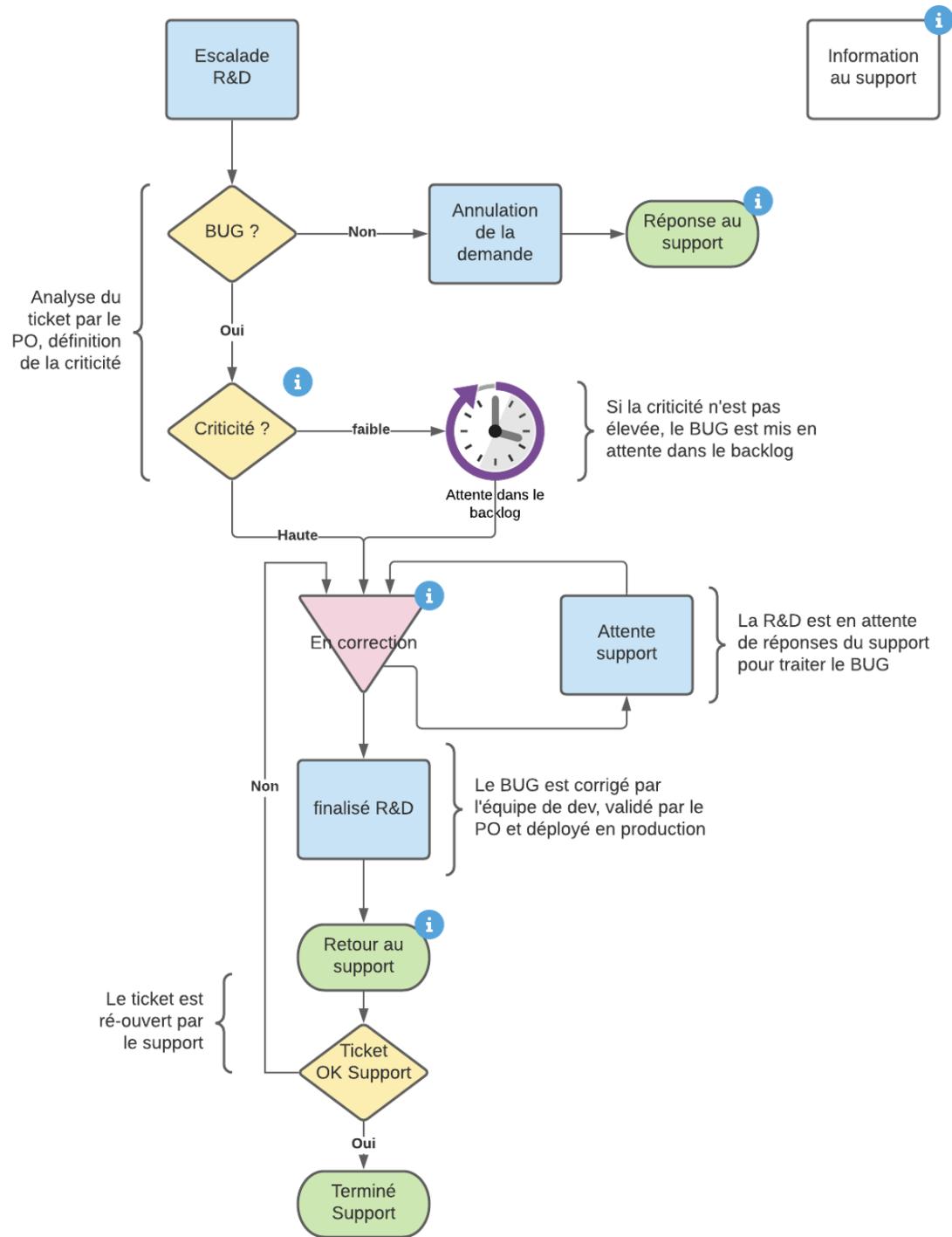


FIGURE A.5 – Diagramme d'activité du traitement des bogues.

A.2 Intégration Continue et Déploiement Continu

Code source A.1 – Le fichier de configuration de Nginx (`conf.d`) du projet Pipeline documentaire utilisé en production.

```

1  server {
2      server_name pp-pipelinedoc.corp.suivideflotte.net;
3      listen          80;
4      return         301 https://pp-pipelinedoc.corp.suivideflotte.net$request_uri;
5  }
6  server {
7      server_name pp-pipelinedoc.corp.suivideflotte.net;
8      listen          443 ssl;
9      ssl_certificate /etc/nginx/certs/fullchain.pem;
10     ssl_certificate_key /etc/nginx/certs/privkey.pem;
11     ssl_protocols    TLSv1 TLSv1.1 TLSv1.2;
12     ssl_ciphers      HIGH:!aNULL:!MD5;
13     index index.php index.html;
14     error_log /var/log/nginx/app-error.log;
15     access_log /var/log/nginx/app-access.log;
16     if ($host = 'www.pp-pipelinedoc.corp.suivideflotte.net' ) {
17         rewrite ^/(.*)$ https://pp-pipelinedoc.corp.suivideflotte.net/$1
18             permanent;
19     }
20     root /var/www/apps/pipelinedoc-source/public;
21     client_max_body_size 25m;
22     location ~ \.php$ {
23         try_files $uri =404;
24         fastcgi_split_path_info ^(.+\.php)(/.+)$;
25         fastcgi_pass pipelinedoc:9000;
26         fastcgi_index index.php;
27         include fastcgi_params;
28         fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
29         fastcgi_param PATH_INFO $fastcgi_path_info;
30     }
31     location / {
32         try_files $uri $uri/ /index.php?$query_string;
33         gzip_static on;
34     }
}

```