

Travail de Bachelor

Information and Communication Technologies (ICT)

Automatisation SaaS de la facturation de Veetamine

Auteur :

Maxime Dénervaud

Professeur :

Prof. Jean-Pierre Rey

Résumé

Le processus de facturation du produit Veetamine est actuellement répétitif et dépend fortement de l'intervention humaine pour l'envoi des factures. Bien que Cellsmaniak, propriétaire de Veetamine, ne soit pas encore sous pression en raison d'un nombre de clients relativement faible, les perspectives de croissance de la start-up sont prometteuses, et une augmentation de la clientèle est attendue d'ici la fin de l'année.

Dans le cadre de la stratégie typique des start-ups, le lancement sur le marché avec un produit minimum viable (MVP) est privilégié pour s'adapter rapidement, attirer des investisseurs, et acquérir les premiers clients. Cette approche permet de générer des revenus plus tôt, mais elle peut aussi poser des défis pour l'évolutivité du produit sur le plan technique. En effet, certains choix techniques et organisationnels précoces, tels qu'une architecture inadaptée, des processus de déploiement manuels en manque de documentation ou une équipe dispersée dans le monde, peuvent entraver la croissance future et l'évolution du produit.

Il a été crucial pour moi de bien comprendre l'état actuel du projet Veetamine afin de répondre aux objectifs fixés et d'évaluer la faisabilité des solutions envisagées. Ainsi, une phase importante de ce projet a été consacrée à l'analyse de l'existant. Cette première étape m'a permis de réviser le Product Backlog et de proposer une solution technique viable pour le projet.

Le projet consiste à développer une API RESTful en .NET, en utilisant le framework ASP.NET Core 8.0. Cette API interagit avec une base de données PostgreSQL pour extraire les données nécessaires. Ces données sont ensuite manipulées avant d'être envoyées à l'API d'Invoice Ninja, l'outil de facturation utilisé par Cellsmaniak. À travers ce projet, plusieurs choix techniques et stratégiques ont été faits, notamment l'adoption d'une nouvelle architecture, la conteneurisation du projet, la mise en place d'une pipeline CI/CD, ainsi que l'intégration d'un services cloud.

Pour finaliser ce projet, une phase d'analyse des résultats a été menée pour vérifier si les objectifs avaient été atteints. Tous les objectifs ont été atteints, bien que le projet soit actuellement isolé du reste de l'infrastructure du produit Veetamine. En effet, une refonte de l'architecture de Veetamine est prévue d'ici la fin de l'année, ce qui permettra d'intégrer pleinement le service d'automatisation de la facturation dans ce nouveau cadre. Cependant, le temps alloué à ce projet m'a permis de développer une solution technique minimale, et des fonctionnalités supplémentaires devront être implémentées pour une intégration complète.

Mots clés : API RESTful, ASP.NET Core, Automatisation de la facturation,

Remerciements

Je souhaite exprimer ma profonde gratitude à toutes les personnes qui m'ont soutenu et aidé dans la réalisation de ce travail de Bachelor :

- **Prof. Jean-Pierre Rey**, pour son encadrement, ses conseils avisés et son soutien tout au long de ce projet.
- **Ubaldo Fiamingo, CTO de Cellsmaniak**, pour m'avoir offert l'opportunité de réaliser ce travail en parallèle de mon immersion professionnelle, enrichissant ainsi mon expérience.
- **The Ark**, pour son soutien à Cellsmaniak, ainsi que pour m'avoir permis de travailler dans un environnement propice à l'innovation.
- **L'équipe de développement de Cellsmaniak**, pour leur accueil chaleureux, leur collaboration et leur disponibilité pour répondre à mes questions.
- **Ma famille et mes amis**, pour leur soutien et leur compréhension tout au long de cette aventure.

Table des matières

Table des matières	iv
Table des figures	ix
Liste des tableaux	xi
1 Introduction	1
1.1 Contexte et problématique	1
1.2 Processus actuel	1
1.3 Compréhension des parties prenantes	2
1.4 Objectifs	3
1.5 Délivrables	3
1.6 Ressources à disposition	4
1.7 Processus souhaité	4
2 Analyse de l'existant	5
2.1 Architecture actuelle du projet	5
2.2 Frontend	5
2.3 Backend	6
2.3.1 Les données de facturation	7
2.3.2 Le lien existant entre les données de facturation	7
2.3.3 Architecture monolithique	9
2.3.4 Environnement de développement	9
2.4 Invoice Ninja	10
2.5 Indentification du projet de bachelor	11
3 État de l'art	12
3.1 Introduction	12
3.2 .NET	12
3.2.1 Introduction .NET	12
3.2.2 Le langage C#	13
3.2.3 Types de projets	13
3.2.4 Outils de développement	13
3.2.5 ASP.NET Core : Contrôleurs/API minimales	14
3.2.5.1 Contrôleurs	14

3.2.5.2	API minimales	14
3.2.5.3	Comment faire son choix ?	15
3.3	Architectures	15
3.3.1	Principes de l'architecture	15
3.3.1.1	Séparation des responsabilités	15
3.3.1.2	Encapsulation	16
3.3.1.3	Inversion des dépendances	17
3.3.2	Application tout-en-un	18
3.3.3	Architecture N-tier	19
3.3.4	Clean Architecture	22
3.3.4.1	Couche Domaine	23
3.3.4.2	Couche Application	23
3.3.4.3	Couche Infrastructure	24
3.3.4.4	Couche Présentation	24
3.4	Docker	24
3.4.1	Conteneurisation VS Virtualisation	24
3.4.2	Docker en production	25
3.4.3	Exécuter une base de données dans un conteneur	26
3.4.4	Est-ce nécessaire de déployer une base de données conteneurisée ?	26
3.4.4.1	Evolutivité	26
3.4.4.2	Facilité d'installation et de configuration	27
3.4.4.3	Portabilité	27
3.4.5	Conclusion	28
4	Choix préalables au développement	29
4.1	Contrôleurs ou API Minimales	29
4.2	L'architecture	29
4.2.1	Comparaison entre l'architecture N-Tier et la Clean Architecture	29
4.2.1.1	Principes de base	29
4.2.1.2	Structure	30
4.2.1.3	Objectifs	30
4.2.1.4	Complexité de la logique métier	30
4.2.1.5	Conclusion	31
4.2.2	Choix de l'architecture	31
4.3	La base de données	31
4.3.1	Mes prérequis	32
4.3.2	Introduction : MySQL et PostgreSQL	32
4.3.3	Comment choisir entre MySQL et PostgreSQL ?	32
4.3.4	Microsoft SQL Server vs PostgreSQL	33
4.3.4.1	Introduction : Microsoft SQL	33
4.3.4.2	Tarification	34
4.3.4.3	Plateformes compatibles	34

Table des matières

4.3.4.4	Syntaxe et langage	34
4.3.5	Choix de la base de données	35
4.4	Service Cloud	35
4.4.1	Qu'est-ce que le PaaS ?	35
4.4.2	Utilisation de ChatGPT 4o pour obtenir des recommandations	35
4.4.3	Choix du service cloud	36
4.5	Conteneuriser la base de données en production	36
5	Méthodologies	38
5.1	Gestion de projet	38
5.1.1	Modèle et choix de gestion de projet	38
5.1.2	Planification du projet	38
5.1.2.1	Sprint 1	39
5.1.2.2	Sprint 2	39
5.1.2.3	Sprint 3	40
5.2	Outils et technologies	40
5.2.1	Environnement de développement	40
5.2.2	Outils de productivités	41
5.2.3	Outils de versionnage	42
5.2.3.1	Workflow de branches	43
5.3	Processus de développement	43
5.3.1	Étape 1 : Analyse et Planification	43
5.3.1.1	Identification des limites du processus actuel	44
5.3.2	Étape 2 : Mise en place de l'environnement de développement	45
5.3.2.1	Assurer la communication entre les conteneurs	45
5.3.3	Étape 3 : Modélisation de la base de données	46
5.3.3.1	Création des entités	46
5.3.3.2	Synchronisation avec la base de données	47
5.3.4	Étape 4 : Mise en place de l'environnement de production	47
5.3.5	Étape 5 : Développement de l'API	48
5.3.5.1	Création de Endpoints selon la Clean Architecture	48
5.3.5.2	Tester l'API de Invoice Ninja	48
5.3.6	Étape 6 : Gestion des données clients	49
5.3.7	Étape 7 : Génération et envoi des factures	49
5.3.7.1	Vérification de l'envoi des factures	50
5.3.8	Étape 8 : Automatisation de la génération des factures	51
5.3.9	Étape 9 : Test et déploiement	52
5.4	Utilisation de l'intelligence artificielle	52
5.4.1	Copilot	52
5.4.2	ChatGPT 4o	53
6	Réalisation	54

6.1	Le service InvoiceGenerationService	54
6.1.1	Méthode GenerateInvoicesAsync	54
6.1.2	Méthode CalculateDates	54
6.1.3	Méthode ProcessOperatorInvoices	55
6.1.4	Méthode CalculateInvoiceValues	55
6.1.5	Méthode PrepareInvoiceDto	56
6.1.6	Méthode GenerateLineItems	57
6.1.7	Méthode SendInvoice	57
6.1.8	Conclusion	57
6.2	Cron Job	57
6.2.1	Persistance des données Quartz.NET	59
6.2.2	Contrôler le bon fonctionnement du Cron Job	59
6.3	Middleware - Rate Limiting	60
6.3.1	OWASP	60
6.3.2	Rate Limiting middleware ASP.NET Core	60
6.3.3	Configuration	61
6.3.4	Implémentation du middleware	61
6.3.5	Contrôler le bon fonctionnement du middleware	62
6.4	Déploiement de l'API Conteneurisé dans Heroku	63
6.4.1	Image Docker	63
6.4.2	Pipeline CI/CD avec GitHub Actions	64
7	Résultats	66
7.1	Mapper les données	66
7.1.1	Résultat du mapping	67
7.2	Génération des factures	68
8	Discussions	71
8.1	Gestion de projet	71
8.1.1	Déroulement du projet	71
8.1.2	Apprentissage	71
8.1.3	Avis	71
8.2	Mappage	72
8.2.1	Avantages	72
8.2.2	Inconvénients	72
8.3	Génération des factures	72
8.3.1	Etat des lieux	72
8.3.2	Axe d'amélioration	72
8.4	Retour d'expérience Clean Architecture	73
8.4.1	Mise en place	73
8.4.2	Structure de l'intégration d'une méthode	74
8.4.3	Avis et recommandations	75

Table des matières

9 Conclusion	76
I Annexes	77
I.1 Méthode GenerateInvoicesAsync	77
I.2 Méthode CalculateDates	77
I.3 Méthode ProcessOperatorInvoices	78
I.4 Méthode CalculateInvoiceValues	79
I.5 Méthode PrepareInvoiceDto	79
I.6 Méthode GenerateLineItems	80
I.7 Méthode SendInvoice	81
I.8 Sprint 1	82
I.9 Sprint 2	83
I.10 Sprint 3	83
Références	84

Table des figures

1.1	Illustration du processus de facturation avant la réalisation du projet	2
1.2	Illustration du processus de facturation souhaité	4
2.1	Illustration de l'architecture actuelle du projet Veetamine	5
2.2	Données de facturation de la table OperatorInvoices	8
3.1	Schéma des dépendances de compilation directe	17
3.2	Schéma des dépendances de compilation inversé	18
3.3	Application tout-en-un, organisation de base fournit par Microsoft	19
3.4	Architecture en N couches conventionnelle	20
3.5	Représentation d'un projet ASP.NET implémentant une architecture N-Tier dans Visual Studio	21
3.6	Clean Architecture	23
3.7	Conteneurisation VS Virtualisation	25
5.1	Tableau blanc numérique Excalidraw	41
5.2	Processus de génération de factures	44
5.3	Modèle physique de données (MPD) de la facturation de Veetamine	46
5.4	Détail de la facture client	49
5.5	Table OperatorInvoices modifiée pour assurer la vérification de l'envoi des factures	51
5.6	Explication de l'expression cron	51
5.7	Auto-complétion de GitHub Copilot	53
6.1	Test du endpoint RateLimited	63
6.2	Étape d'intégration dans le pipeline	65
6.3	Étape de déploiement dans le pipeline	65
7.1	Mise à jour de l'opérateur depuis le dashboard de Veetamine	66
7.2	Point de terminaison pour attribuer une personne de référence	67
7.3	Schéma de données de "Opérateur" prouvant le mappage	67
7.4	Détail d'un client depuis l'interface de Invoice Ninja	68
7.5	Point de terminaison pour déclencher le Cron Job manuellement	69
7.6	Listes de factures depuis Invoice Ninja	69
7.7	Détail d'une facture en PDF pour un client	70
I.1	Sprint 1 - Azure DevOps	82

Table des figures

I.2	Sprint 2 - Azure DevOps	83
I.3	Sprint 3 - Azure DevOps	83

Liste des tableaux

2.1	Détail d'une facture envoyée au début du mois d'avril	8
6.1	Tableau des requêtes disponibles, prises, recyclées et reportées au fil du temps . .	62

1 | Introduction

1.1 Contexte et problématique

Cette thèse de Bachelor est conduite en parallèle de mon immersion professionnelle chez Cellsmaniak SA, une entreprise IT située à Sion, en Valais. C'est en collaboration avec cette entreprise que je vais mener à bien ce projet de recherche. Actuellement, Cellsmaniak est en cours de développement d'un nouveau produit dénommé Veetamine (VTM). Veetamine est une plateforme SaaS (Software-as-a-Service) conçue pour moderniser le secteur des distributeurs automatique (Vending). Le projet, lancé en 2021, se trouve à l'étape de start-up. À ce jour Cellsmaniak a conclu un accord commercial avec un opérateur et est en pourparlers avec de potentiels nouveaux clients. Dans la situation actuelle, la facturation ne présente pas un défi majeur, l'entreprise n'ayant qu'un seul client.

Toutefois, le véritable enjeu se présentera lorsque Cellsmaniak élargira sa clientèle. Actuellement, le processus de facturation n'est pas automatisé. Au début de chaque mois, une personne responsable est chargée de saisir manuellement les données de facturation dans Invoice Ninja, le logiciel utilisé par l'entreprise. Cette méthode est susceptible d'introduire des erreurs humaines et résulte en un processus lent et fastidieux.

1.2 Processus actuel

L'illustration ci-dessous montre les différentes étapes où une intervention humaine est nécessaire pour l'envoi des factures. Aujourd'hui, cette tâche représente environ 30 minutes de travail pour l'envoi d'une facture.

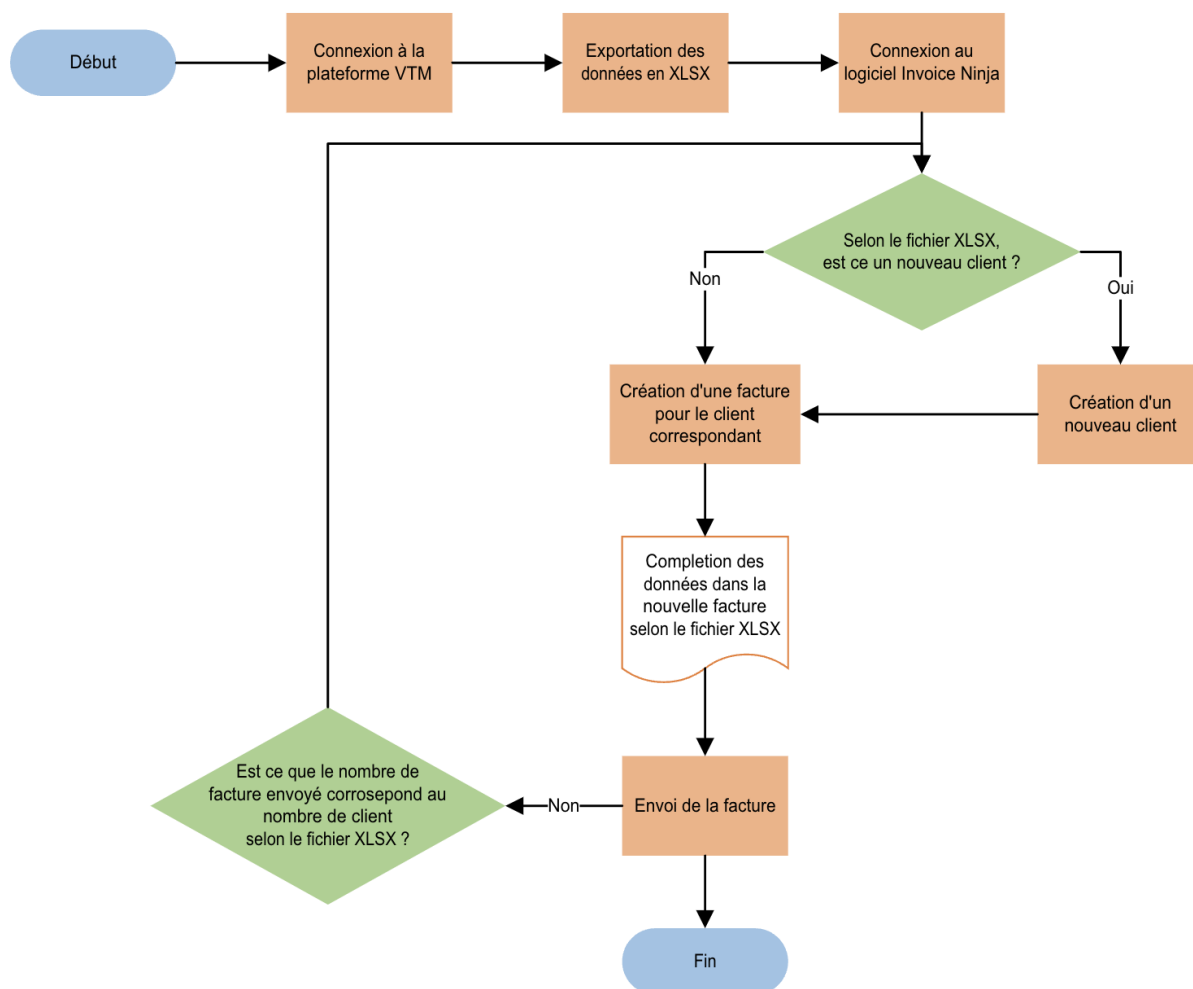


FIGURE 1.1 – Illustration du processus de facturation avant la réalisation du projet
Source: de l'auteur

1.3 Compréhension des parties prenantes

Cellsmaniak SA, une société anonyme établie en Valais à l'Avenue de Tourbillon 5, Sion, bénéficie du soutien de la fondation The Ark, qui fournit un service d'incubation pour start-ups. Ce soutien est complété par l'encadrement de Cimark, qui propose un accompagnement personnalisé aux start-ups, incluant la mise à disposition d'infrastructures telles que des bureaux, des espaces de coworking, des événements de réseautage, ainsi qu'une aide active dans la recherche de financements. L'équipe de Cellsmaniak se compose exclusivement de professionnels seniors ayant chacun plus de 25 ans d'expérience. ARK, 2024

Veetamine, le produit de Cellsmaniak SA, est une solution SaaS destinée à la gestion de machines automatiques de différentes marques. Elle se caractérise par son intégration fluide, ses économies de coûts et sa flexibilité, permettant la gestion de données bidirectionnelles telles que les ventes, la maintenance, et les mises à jour de firmware, sans nécessiter de matériel

de connectivité supplémentaire. Veetamine se veut neutre, accessible à tous les intervenants du secteur et ambitionne de moderniser l'industrie de la distribution automatique. VEETAMINE, 2024a

Fabricants, constructeurs de machines automatiques, désormais cinq fabricants peuvent être connectés à Veetamine, notamment Franke, Eversys, Rancilio, Fas et Reha. VEETAMINE, 2024b

Clientèle de Veetamine se décline en deux niveaux actuels, avec la possibilité d'ajouter des catégories supplémentaires au fur et à mesure de l'évolution du produit. Le premier niveau concerne les opérateurs, fournisseurs de machines qui collaborent avec plusieurs fabricants. Ces opérateurs déploient les machines dans différentes enseignes, constituant ainsi le deuxième niveau de la clientèle. Un exemple concret est la collaboration de Veetamine avec l'opérateur Dallmayr, qui équipe Coop Pronto en machines. VEETAMINE, 2024b

1.4 Objectifs

L'objectif principal de ce projet est de synchroniser les données de facturation de Veetamine avec le logiciel Invoice Ninja, en utilisant les API RESTful des deux plateformes. Cette intégration permettra de générer automatiquement des factures dans Invoice Ninja à partir des données de Veetamine, rationalisant ainsi le processus de facturation pour gagner du temps et améliorer la qualité des opérations.

Le projet inclura également la mise en place d'un pipeline d'intégration continue (CI) et de déploiement continu (CD) pour le backend de Veetamine. L'objectif principal de cette initiative est d'améliorer l'efficacité et la rapidité des déploiements, tout en assurant une meilleure qualité du code.

1.5 Délivrables

Les livrables de ce projet sont :

- Une génération automatique des factures dans Invoice Ninja à partir des données de Veetamine.
- L'intégration entre l'API de Veetamine et l'API de Invoice Ninja.
- Un pipeline d'intégration continue et de déploiement continu.

1.6 Ressources à disposition

J'ai accès aux dépôts GitHub pour les applications frontend et backend, ainsi qu'à la base de données. Je dispose également de certains droits administrateurs sur le compte Invoice Ninja de l'entreprise. En cas de besoin, je suis en mesure de solliciter et d'obtenir des droits supplémentaires. Je peux également demander de débloquer un budget financier si cela est justifiable.

En termes d'encadrement, je travaillerai seul sur le projet. Cependant, une à deux fois par semaine, je compte communiquer sur l'avancement du projet et les choix que j'entreprends avec l'équipe de développement. Jean-Pierre Rey joue également un rôle d'encadrement, mais avec une orientation plus académique.

1.7 Processus souhaité

L'illustration ci-dessous présente le processus de facturation souhaité sans une étude approfondie de faisabilité basée sur l'existant. On peut y voir que la gestion des clients serait effectuée via Invoice Ninja et non depuis Veetamine. La création des clients serait réalisée manuellement, tout comme l'envoi final des factures.

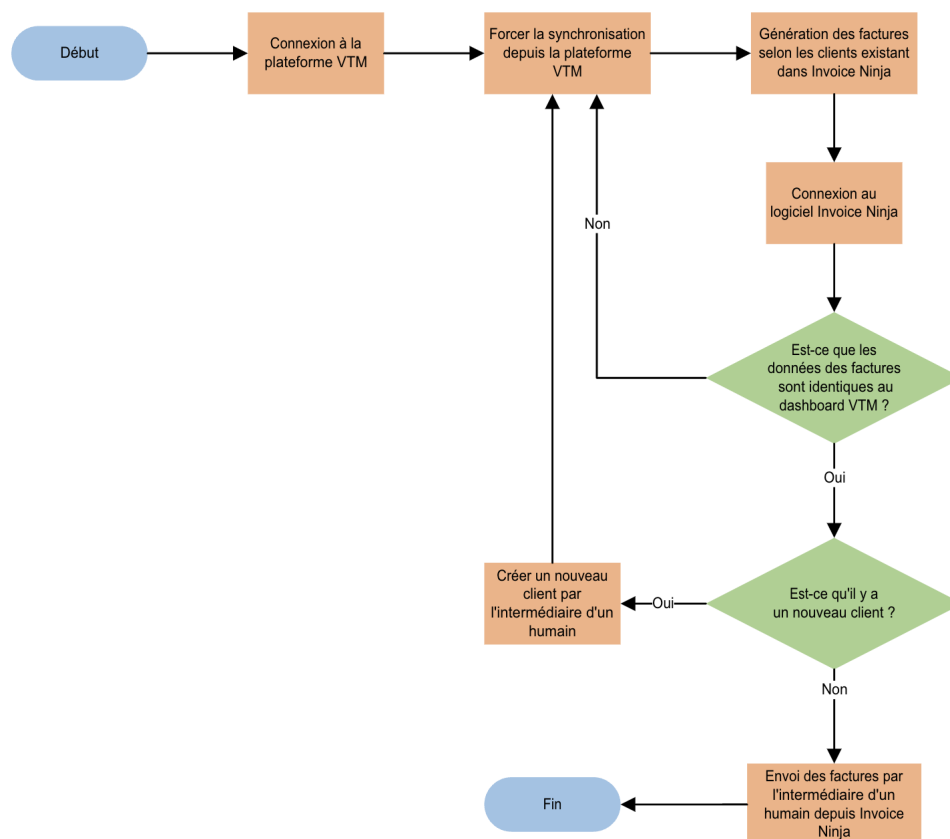


FIGURE 1.2 – Illustration du processus de facturation souhaité
Source: de l'auteur

2 | Analyse de l'existant

2.1 Architecture actuelle du projet

Dans cette section, je vais décrire l'état actuel du projet Veetamine dans son ensemble. L'objectif est de fournir un maximum de contexte pour me permettre de prendre les décisions les plus appropriées concernant l'intégration de mon projet. Par la même occasion, je pourrai identifier certaines interrogations qui pourront trouver des réponses dans la partie "état de l'art" de ce travail.

k

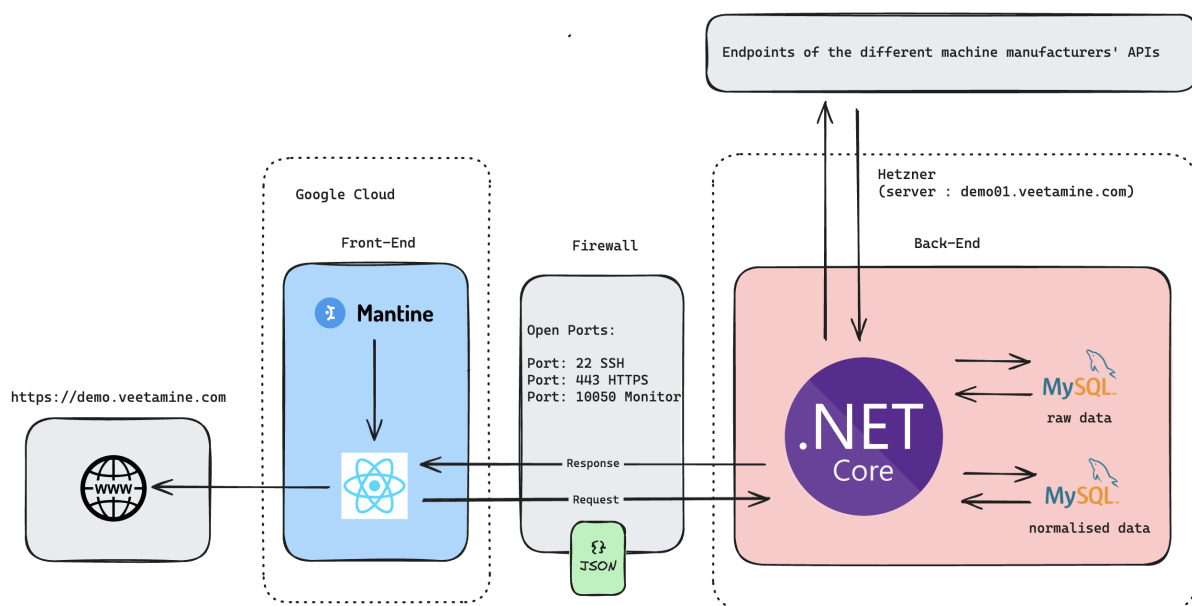


FIGURE 2.1 – Illustration de l'architecture actuelle du projet Veetamine
Source: de l'auteur

2.2 Frontend

Pour l'environnement frontend l'équipe de Cellsmaniak a fait le choix d'utiliser la bibliothèque React JS combiné avec TypeScript et d'utiliser Vite.js comme bundler. Vite est un outil de bundling qui permet de générer une version optimisée et prête à l'emploi du projet lorsque la commande "npm run build" est utilisée. RADIX, 2024

React JS est une bibliothèque JavaScript open source populaire développée par Meta, anciennement Facebook. Elle a été introduite pour la première fois en 2013. Aujourd'hui, React est maintenu par Meta ainsi par une communauté de développeurs individuels et des entreprises. Il est particulièrement adapté pour le développement frontend, permettant de créer des interfaces utilisateur interactives pour les applications web monopages (SPA) et pour des applications mobiles. for GEEKS, 2024 ; WIKIPEDIA, 2024b

TypeScript est un sous-ensemble de JavaScript qui ajoute des fonctionnalités de typage statique au langage. Cela permet de détecter les erreurs de code lors du développement, rendant le processus plus fiable et facilitant la maintenance à long terme. En d'autres termes, TypeScript aide les développeurs à écrire du code plus propre et plus robuste. KINSTA, 2023

Voici ci-dessous une liste non exhaustive des bibliothèques et librairies utilisées pour le développement frontend :

- **Mantine UI** pour les composants React personnalisables
- **Tanstack** pour la gestion des états et des données
- **Axios** pour les requêtes HTTP
- **Date.js** pour la gestion des dates
- **Rechart** pour les graphiques et les visualisations des données

Cellsmaniak est en cours de migration d'Infomaniak vers Google Cloud Platform (GCP) comme principal fournisseur de services cloud. Actuellement, seule la version de démonstration du frontend est hébergée sur GCP. Bien que je ne connaisse pas les raisons exactes de ce choix, Google offre actuellement la meilleure solution pour leurs besoins. Ils en profitent également pour intégrer une pipeline CI/CD afin d'automatiser le déploiement de leur application frontend, remplaçant ainsi l'utilisation d'un gestionnaire FTP tel que FileZilla qu'ils utilisaient auparavant pour Infomaniak.

2.3 Backend

Concernant le backend, Veetamine est une API RESTful développée avec le framework ASP.NET Core WebAPI, basée sur .NET 8. L'API interagit avec plusieurs services tiers, notamment des fabricants de machines automatiques, en utilisant la méthode de polling pour la communication. Cette méthode consiste à envoyer des requêtes HTTP à intervalles réguliers, par exemple toutes les 5 minutes, pour vérifier les mises à jour des données. Bien que simple à mettre en œuvre, cette méthode est inefficace en termes de performances et de consommation de ressources et ne permet pas des mises à jour en temps réel. SVIX, s. d.

Pour remédier à ces limitations, Cellsmaniak envisage d'adopter les WebSockets à l'avenir. Les WebSockets offrent une communication bidirectionnelle et permettent d'établir une connexion persistante entre le client et le serveur, facilitant ainsi les mises à jour en temps réel. MOZILLA, s. d.

Les données récupérées sont initialement stockées dans une base de données MySQL, organisées par intégration. Elles sont ensuite transformées et standardisées pour être stockées dans une seconde base de données MySQL. Enfin, ces données sont exposées via l'API RESTful et peuvent être utilisées par le frontend. La documentation de l'API est réalisée en utilisant OpenAPI pour la spécification. OpenAPI fournit une description détaillée du fonctionnement de l'API à travers un fichier au format JSON ou YAML. Swagger est ensuite utilisé pour générer une documentation interactive, permettant aux développeurs de comprendre et de tester facilement l'API. GRAFIKART, 2019

MySQL est actuellement le système de gestion de base de données relationnelle utilisé. Cependant, pour mon projet de Bachelor, j'ai la possibilité d'explorer d'autres solutions de bases de données.

À l'avenir, le backend sera déployé sur Google Cloud Platform (GCP), tout comme le frontend. Pour l'instant, le backend est hébergé chez Hetzner, un fournisseur bien connu de longue date pour l'entreprise.

2.3.1 Les données de facturation

Pour mon travail de Bachelor, les données sont la source principale pour que je puisse générer des factures. Il est donc important de d'abord vérifier si elles sont disponibles, puis de comprendre comment les interpréter. La table principale des données de facturation se nomme "OperatorInvoices". En inspectant les données, j'ai identifié qu'elles étaient sauvegardées à chaque début de mois et que deux lignes étaient créées par opérateur (client). Je reviendrai sur ce point plus tard. Cependant, selon le Product Backlog initial du projet et le processus souhaité réalisé au point 1.7, je devais implémenter une user story permettant de forcer la synchronisation des données depuis le frontend.

Selon moi puis avec validation d'un collègue, il est inutile de forcer une synchronisation, sachant que les dernières données disponibles ne seront accessibles qu'au début de chaque mois. Par conséquent, cette user story a été supprimée du Product Backlog.

2.3.2 Le lien existant entre les données de facturation

En découvrant la table "OperatorInvoices", j'ai ressenti le besoin de vérifier et de comprendre comment le lien était établi entre les données de facturation présentes dans l'onglet "VTM report" du dashboard Veetamine et celles de la base de données. Si aucun lien n'est établi, cela signifie que je devrai poursuivre mon analyse dans le code source de l'application pour comprendre la relation. Si tel est le cas, cela orienterait mon travail de Bachelor dans une autre direction. Il est donc nécessaire d'élucider ce point dans la partie analyse de l'existant, plutôt que dans la méthodologie de mon travail de Bachelor, bien que cela en fasse également partie.

Chapitre 2. Analyse de l'existant

Pour le comprendre, je suis parti d'une facture existante envoyée au début du mois d'avril. L'illustration ci-dessous présente le détail de cette facture, incluant les quatre services facturés pour chaque client ainsi que son montant total hors TVA (qui est de 8,1 %). À titre d'information, les coûts unitaires sont définis selon un contrat, et les quantités sont calculées par un algorithme qui prend en compte le nombre de machines connectées et les données de transactions.

Parallèlement, j'ai inspecté la table "OperatorInvoices". Les données encadrées en rouge regroupent les informations nécessaires pour le calcul et la génération de la facture du mois d'avril.

TABLE 2.1 – *Détail d'une facture envoyée au début du mois d'avril*

Description	Unit Cost	Quantity	Line Total
Fix fee per machine per month (Operator)	7.00	451	CHF 3'157.00
Variable fee per machine per month (Operator)	0.0036	547'160.00	CHF 1'969.78
Fix fee per machine per month (VTM)	1.00	451.00	CHF 451.00
Variable fee per machine per month (VTM)	0.0024	547'160.00	CHF 1'313.18
Total	-	-	CHF 6'890.96

Id	OperatorId	ContractId	MachinesCount	TransactionCount	TransactionUsed	MachinesFees	TransactionFees	Year	Month	CalculatedAt
41	2	3	464	516422	555604	464	1333.4496	2024	4	2024-05-01 00:00:11.215160
40	2	3	457	522388	549388	457	1318.5312	2024	3	2024-05-01 00:00:11.215160
39	2	3	458	514805	547196	458	1313.2703999999999	2024	3	2024-04-01 00:15:14.798427
38	2	3	452	517103	544130	452	1305.9119999999998	2024	2	2024-04-01 00:15:14.798427
37	2	3	459	504851	544166	459	1305.9984	2024	2	2024-03-01 00:21:18.696179

FIGURE 2.2 – *Données de facturation de la table OperatorInvoices*
Source: de l'auteur

À première vue, il n'est pas évident de comprendre le lien entre les données de facturation et les données de la base de données. Voici une explication détaillée :

Au début du mois courant, mais dans mon explication du mois d'avril, le code source de l'API de Veetamine sauvegarde deux lignes (Id : 39 et Id : 38) par client dans la base de données. Nous pouvons le constater grâce à la colonne "CalculatedAt". La première ligne (Id : 38) est une recalculation des données de deux mois plus tôt (février) et la seconde ligne (Id : 39) est une calculation des données du mois précédent (mars). Cette particularité est due au fait que, durant le mois, certaines machines peuvent tomber en panne et, par conséquent, ne plus générer de coûts pour Veetamine. Le véritable problème réside dans le fait que certaines machines connectées intègrent des technologies obsolètes des années 70, ce qui fait que les données ne sont pas toujours à jour le premier jour du mois lors de la facturation. La solution actuelle consiste à rattraper ce délai d'informations à les ajouter ou les soustraire de prochaine facture du client. C'est pour cette raison qu'il y a des données des mois de février et mars dans la facture du mois d'avril.

Pour la suite de la compréhension, les deux colonnes qui m'intéressent pour la génération de la facture sont "TransactionUsed" et "MachineFees". Toutefois, la logique est identique pour les deux colonnes, donc je vais prendre "MachineFees" pour l'exemple.

Pour arriver à la quantité de 451 pour "Fix fee per machine per month" dans la colonne "MachineFees", je dois prendre la valeur de la colonne "MachineFees" du mois précédent (mars), qui est de 458, et y ajouter la différence entre les deux valeurs de la colonne "MachineFees" des données de deux mois plus tôt (février), soit (452 - 459), en respectant l'ordre d'ajout dans la base de données. Pour simplifier l'explication, voici le détail du calcul :

Fix fee per machine per month :

$$451 = 458 + (452 - 459)$$

Variable fee per machine per month :

$$547'160 = 547'196 + (544'130 - 544'166)$$

2.3.3 Architecture monolithique

L'API de Veetamine repose sur une architecture monolithique, ce qui signifie que toutes les fonctionnalités sont regroupées dans un seul projet. Cette approche facilite la gestion du code et offre une compréhension globale de l'application, tant que la taille du projet reste raisonnable. Elle permet également une simplicité de développement et de déploiement. Étant donné que Veetamine est en phase de start-up, ce choix d'architecture se justifie car il leur a permis de proposer rapidement un MVP (Minimum Viable Product) et de tester leur produit sur le marché.

Cependant, l'architecture monolithique présente des limites, notamment en termes de scalabilité. Aujourd'hui, Veetamine est connecté à environ 500 machines, ce qui reste gérable. Cellsmaniak peut encore procéder à des mises à l'échelle verticale de leur infrastructure, c'est-à-dire augmenter la capacité de la machine en ajoutant plus de CPU ou de mémoire, mais cela a un coût. Il est évident qu'il est impossible de se reposer indéfiniment sur cette méthode en raison des limites physiques. C'est pourquoi Cellsmaniak envisage, dans le dernier trimestre de cette année, de mettre en œuvre une nouvelle architecture. L'architecture microservices, qui se décompose en plusieurs projets, pourrait être une solution. PERRAUDEAU, 2021 ; POCHELET, 2024

2.3.4 Environnement de développement

Initialement, le projet a été construit sur les connaissances d'un seul développeur. Aujourd'hui, le projet devient de plus en plus complexe et l'intégration de nouvelles fonctionnalités prendrait beaucoup plus de temps si elle était réalisée par un autre développeur. De plus, l'environnement

de développement n'est pas encore partageable. Une solution serait de conteneuriser le projet avec Docker, mais là encore, un seul développeur possède les connaissances du projet pour le faire.

Dans cette situation, mes options pour la réalisation de mon projet de Bachelor sont limitées. Sans un environnement de développement partageable, je ne pourrai pas travailler sur le backend. C'est pourquoi la meilleure solution est de créer un nouvel environnement partageable tout en respectant certaines contraintes de Veetamine. De plus, cela permettra d'acquérir de l'expérience pour la nouvelle architecture que Cellsmaniak envisage de mettre en place.

2.4 Invoice Ninja

Invoice Ninja fait également partie de l'infrastructure existante. C'est le nouveau service tiers que je dois intégrer à l'écosystème de Veetamine.

Invoice Ninja est une application de facturation qui simplifie et facilite l'envoi de factures et la réception de paiements. Invoice Ninja est une petite équipe de trois fondateurs : Hillel, Dave et Shalom, qui ont lancé leur premier service en ligne en 2014. Dix ans plus tard, le projet est toujours maintenu et s'autofinance, c'est-à-dire sans financement externe, sans capital risque. NINJA, 2024b

Le projet est open source et offre la possibilité d'héberger Invoice Ninja sur les serveurs de son choix. Une autre option consiste à payer un abonnement annuel pour que Invoice Ninja s'occupe de l'hébergement.

Dans le cas de Cellsmaniak, ils ont opté pour les services d'Infomaniak pour l'hébergement. Toutefois, Cellsmaniak doit s'acquitter d'une licence de 30 dollars par an s'ils souhaitent retirer le logo de Invoice Ninja qui est visible par les clients. NINJA, 2024d

Aujourd'hui, la version 5 de l'application utilise le framework PHP Laravel pour le code source de l'API et pour l'environnement backend. Du côté frontend, partie portail d'administration ordinateur et mobile, c'est Flutter qui est utilisé. Flutter est un framework open source de Google permettant de créer des applications multiplateformes compilées nativement à partir d'une seule base de code. En effet, Invoice Ninja est compatible avec Android, iOS, macOS, Linux et Windows. Puis, pour l'utilisation de l'application dans le web, c'est React JS qui est choisi. AWS, s. d.

De plus, Ninja mentionne dans leur documentation d'utiliser plutôt la version web pour les entreprises qui disposent de beaucoup d'utilisateurs pour des raisons de performances. En effet, les applications pour mobile et desktop chargent toutes les données lors de la première connexion, ce qui rend la navigation très rapide, mais contre-productive lorsqu'il y a beaucoup d'utilisateurs. Contrairement à l'application web où les données sont chargées à la demande. NINJA, 2024c

2.5 Identification du projet de bachelor

Grâce à cette analyse de l'existant, j'ai une meilleure compréhension de mon projet de Bachelor. De plus, je peux maintenant identifier mes axes de recherche et d'analyse pour la phase de l'état de l'art. Voici une redéfinition de mon projet de Bachelor :

Je devrai partir d'un nouveau projet ASP.NET Core WebAPI en respectant la version actuelle de Veetamine. Cet environnement devra être facilement partageable et temporairement déployable sur un fournisseur cloud de mon choix. Les services tels qu'Amazon Web Services (AWS), Microsoft Azure ou Google Cloud Platform (GCP) sont à éviter en raison de leur complexité et de leurs coûts, car ce n'est pas ce qui est attendu de mon projet. L'objectif principal est plutôt la génération automatique des factures dans Invoice Ninja à partir des données de Veetamine.

Mon application devra récupérer des données provenant d'une base de données de mon choix, mais avec une modélisation similaire à celle de l'existant, en respectant les noms de tables et de colonnes appropriés. Ces données seront manipulées afin de générer des factures qui seront envoyées à Invoice Ninja au début de chaque mois. Enfin, je ne devrai pas me préoccuper du déploiement d'une nouvelle instance d'Invoice Ninja car il est possible de créer une nouvelle entreprise depuis l'application existante.

3 | État de l'art

3.1 Introduction

Trois axes de recherche me conduisent à explorer cette partie de l'état de l'art. Le premier axe concerne la technologie .NET. Jusqu'à présent, je n'ai jamais eu l'occasion de travailler avec cette technologie et je souhaite en comprendre le contexte général, en mettant l'accent sur le développement d'API. Le deuxième axe se focalise sur l'architecture. En effet, je me demande s'il existe une architecture recommandée pour le développement d'une API avec .NET afin de garantir une application évolutive. Enfin, le troisième axe concerne Docker. Bien que j'aie déjà utilisé Docker, je n'ai jamais poussé son utilisation jusqu'à la phase de production pour un projet réel. Je souhaite donc comprendre comment Docker est utilisé en production, notamment en ce qui concerne la gestion des bases de données.

3.2 .NET

3.2.1 Introduction .NET

.NET est né sous la forme de .NET Framework, avec une architecture fortement couplée à Windows. Il a vu le jour dans un contexte où Java dominait le marché et où la guerre entre plateformes était très polarisée. .NET a conservé cette philosophie tout au long de l'évolution de la famille .NET Framework, jusqu'à la version 4.8 (mai 2019). Le framework a rapidement évolué pour adresser différents types de projets (Web, desktop, services Web, services Windows, etc.), ce qui en a fait un des éléments distinctifs de .NET.

En 2016, Microsoft a lancé .NET Core, une version du framework conçue pour être multiplateforme et indépendante de Windows. Ce changement a marqué une évolution significative dans la stratégie de Microsoft, adoptant une approche plus ouverte.

En 2020, un moment clé a été l'unification de .NET Framework et .NET Core sous .NET 5. Les équipes d'ingénierie ont réalisé un travail remarquable pour surmonter les nombreux défis associés à cette unification. Aujourd'hui, nous avons atteint la version 8, caractérisée par une maturité élevée dans la plupart des domaines couverts, garantissant une stabilité, une sécurité et des performances accrues. INZA, 2024

3.2.2 Le langage C#

Microsoft propose trois langages sur la plateforme .NET : C#, F#, et Visual Basic. Parmi eux, C# est le plus populaire et le plus utilisé. C# est un langage de programmation orienté objet, fortement typé et compilé, offrant des fonctionnalités modernes et avancées. Il permet de développer des applications web, mobiles, de bureau et des jeux vidéo de haute qualité. L. MICROSOFT, 2024

3.2.3 Types de projets

Microsoft veille à ce que la plateforme .NET facilite le développement de divers types de projets (liste non exhaustive) INZA, 2024 :

- Développement d'API (ASP.NET WebAPI, minimal API, CoreWCF, etc.)
- Web (Blazor, ASP.NET)
- Desktop (WPF, WinForms, UWP)
- Mobile (MAUI)
- Accès aux données (Entity Framework)
- Développement Azure (SDK Azure)
- Tests unitaires (MSTest, xUnit, NUnit)
- et bien plus encore

3.2.4 Outils de développement

Les outils de développement jouent un rôle crucial dans l'adoption des technologies sous-jacentes. Pour .NET, Visual Studio est sans doute l'un des meilleurs IDE du marché. Il est riche et polyvalent, offrant une expérience de développement difficile à égaler. Visual Studio est principalement disponible pour Windows (bien que des versions pour macOS aient été publiées, elles n'ont pas rencontré un grand succès) et se décline en différentes versions (payantes et gratuites).

Visual Studio Code (VS Code) est l'éditeur multiplateforme proposé par Microsoft. Cet outil est intéressant pour certains usages et peut être enrichi par de nombreuses extensions.

La comparaison entre les deux est inévitable. Sans entrer dans les détails, ils ne sont pas vraiment comparables. En termes absolus, il est difficile de placer VS Code au même niveau que Visual Studio. Ce dernier reste nettement supérieur, ne serait-ce que par la nature du produit, les fonctionnalités et la valeur ajoutée qu'il propose.

Rider de JetBrains est également une excellente alternative. C'est un IDE plus proche de Visual Studio que de VS Code, à la différence près qu'il est multiplateforme. JetBrains est réputé pour développer des extensions et des produits de haute qualité pour les développeurs. INZA, 2024

3.2.5 ASP.NET Core : Contrôleurs/API minimales

Avec ASP.NET Core, il existe deux principales approches pour créer des API : l'utilisation de contrôleurs (Controllers) et les API minimales (Minimal APIs).

3.2.5.1 Contrôleurs

Les contrôleurs représentent la méthode classique pour créer des API dans ASP.NET Core. Ils s'appuient sur le modèle MVC (Model-View-Controller), une architecture qui sépare les préoccupations en trois parties distinctes :

- **Modèle** : Gère les données de l'application.
- **Vue** : Gère la présentation des données.
- **Contrôleur** : Gère la logique métier et les requêtes HTTP.

Les contrôleurs sont responsables de la gestion des requêtes HTTP entrantes. Lorsqu'une requête HTTP est reçue, le contrôleur identifie la méthode appropriée pour traiter cette requête, appelée méthode d'action. Cette méthode d'action traite la requête, effectue la logique métier nécessaire, et renvoie une réponse HTTP au client.

Voici un exemple de contrôleur simple qui traite la méthode d'action GET pour renvoyer un message "Hello world" lorsqu'on accède à l'API :

```
1      using Microsoft.AspNetCore.Mvc;  
2  
3      [ApiController]  
4      [Route("api/[controller]")]  
5      public class HelloController : ControllerBase  
6      {  
7          [HttpGet]  
8          public IActionResult Get()  
9          {  
10             return Ok("Hello World!");  
11          }  
12      }
```

Cette approche est particulièrement adaptée aux applications complexes qui nécessitent une gestion fine des requêtes et des réponses, ainsi qu'une organisation claire du code. KUMAR, 2023 ; SIVA, 2023

3.2.5.2 API minimales

Les API minimales ont été introduites dans ASP.NET Core 6 comme une nouvelle façon de créer des API avec un minimum de dépendances et de configurations. Cette approche est conçue pour être simple et légère, ce qui en fait un choix idéal pour les microservices et les applications qui nécessitent uniquement les composants essentiels.

Avec les API minimales, il est possible de définir les points de terminaison et leurs comportements de manière directe, ce qui rend le développement plus rapide, surtout pour les cas d'utilisation simples ou les prototypes. Cette approche permet de créer des API rapidement, sans avoir à configurer les contrôleurs et autres composants complexes du modèle MVC.

Voici un exemple de code, situé dans le fichier "Program.cs", qui montre comment créer un point de terminaison répondant "Hello World" lorsqu'on accède au chemin racine KUMAR, 2023 ; SIVA, 2023 :

```
1 var builder = WebApplication.CreateBuilder(args);  
2 var app = builder.Build();  
3 app.MapGet("/", () => "Hello World!");  
4 app.Run();
```

3.2.5.3 Comment faire son choix ?

Le choix entre l'utilisation des contrôleurs et des API minimales repose sur plusieurs facteurs KUMAR, 2023 :

- **Taille du projet** : Les contrôleurs sont généralement plus adaptés aux projets de grande envergure avec des architectures complexes, car ils offrent une meilleure organisation et une structure plus robuste.
- **Vitesse de développement** : Les API minimales sont idéales lorsque la rapidité de développement est essentielle, comme pour le prototypage rapide ou les projets de moindre envergure.
- **Compétences de l'équipe** : Il est important de prendre en compte la familiarité de votre équipe avec le modèle MVC par rapport à sa disposition à adopter les API minimales.

3.3 Architectures

3.3.1 Principes de l'architecture

Selon Microsoft, lors de la conception de logiciels, il est crucial de penser à leur facilité de maintenance. Voici quelques principes qui peuvent aider à créer des applications bien organisées et faciles à entretenir. Ces principes recommandent de créer des composants indépendants qui communiquent entre eux via des interfaces. L. MICROSOFT, 2023b

3.3.1.1 Séparation des responsabilités

Un principe à suivre lors du développement est la séparation des responsabilités. Selon ce principe, le logiciel doit être divisé en fonction des types de tâches qu'il effectue. Par exemple, une classe "OperatorService" pourrait être responsable de la gestion des Operateur, tandis qu'une classe "InvoiceGenerationService" pourrait être responsable à la génération des factures.

Chapitre 3. État de l'art

Cette division permet de mieux organiser le code, de le rendre plus compréhensible et de faciliter sa maintenance. Ce principe est directement lié au "Single Responsibility Principle" (SRP) du design pattern SOLID. L. MICROSOFT, 2023b

3.3.1.2 Encapsulation

L'encapsulation est l'un des quatre piliers de la programmation orientée objet. Elle consiste à regrouper les données et les méthodes qui les manipulent au sein d'une même entité, appelée classe. L'objectif principal de l'encapsulation est de masquer les détails d'implémentation et de protéger les données en les rendant privées. L. MICROSOFT, 2023b

En utilisant correctement l'encapsulation, on peut atteindre plusieurs objectifs importants. Le point ci-dessous est illustré par un exemple de code du projet de bachelor :

Isolation des composants : Chaque partie d'une application (composant ou couche) peut modifier son fonctionnement interne sans perturber les autres parties, tant que l'interface publique reste inchangée.

```
1 // SettingInvoiceNinjaService est isolé de l'implémentation de
  ↳ ISettingInvoiceNinjaRepository.
2
3 public class SettingInvoiceNinjaService : ISettingInvoiceNinjaService
4 {
5     private readonly ISettingInvoiceNinjaRepository _repository;
6
7     public SettingInvoiceNinjaService(ISettingInvoiceNinjaRepository repository)
8     {
9         _repository = repository;
10    }
11
12    // ...
13 }
14
```

Accès contrôlé : Les données internes d'un composant ne peuvent être accessibles qu'à travers des méthodes spécifiques (comme des getters et des setters), empêchant tout accès direct.

```
1 // La méthode GetSettingAsync contrôle l'accès
2
3 public async Task<string> GetSettingAsync(string key)
4 {
5     var setting = await _repository.GetSettingByKeyAsync(key);
6     return setting?.Value;
7 }
8
```

Interfaces publiques : Les composants doivent fournir des interfaces bien définies et claires pour interagir avec les autres parties de l'application, facilitant ainsi la communication entre les différentes parties.

```

1 // ISettingInvoiceNinjaService définit les méthodes publiques que SettingInvoiceNinjaService
  ↳ doit implémenter
2 public interface ISettingInvoiceNinjaService
3 {
4     Task<string> GetNextInvoiceNumberAsync();
5     Task<string> GetSettingAsync(string key);
6     Task SetSettingAsync(string key, string value);
7 }
8

```

Couplage faible : En minimisant les dépendances entre les composants, l'application devient plus modulaire et plus facile à maintenir. Les modifications peuvent être effectuées de manière isolée sans impact majeur sur le reste du système.

```

1 // SettingInvoiceNinjaService dépend de l'interface ISettingInvoiceNinjaRepository,
  ↳ favorisant un couplage faible.
2
3 public SettingInvoiceNinjaService(ISettingInvoiceNinjaRepository repository)
4 {
5     _repository = repository;
6 }
7

```

3.3.1.3 Inversion des dépendances

La plupart des applications sont écrites de manière à ce que les dépendances de compilation suivent le flux d'exécution de manière directe. Cela signifie que si la classe A appelle une méthode de la classe B, et que la classe B appelle une méthode de la classe C, alors, lors de la compilation, la classe A dépendra de la classe B, et la classe B dépendra de la classe C.

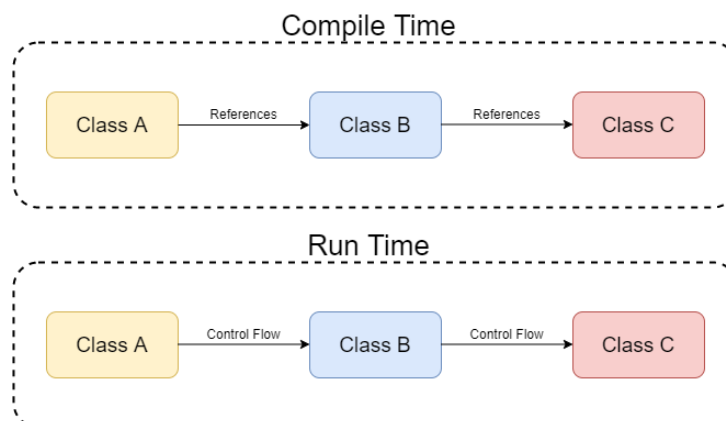


FIGURE 3.1 – Schéma des dépendances de compilation directe
Source: <https://code-maze.com/wp-content/uploads/2020/08/flow.png>

Pour y remédier, le principe d'inversion des dépendances propose que la classe A appelle des méthodes à travers une abstraction, c'est-à-dire une interface, qui est implémentée par la classe B. Ainsi, au moment de l'exécution, A peut utiliser B. Cependant, lors de la compilation, c'est B qui dépend de l'interface contrôlée par A, inversant ainsi la dépendance habituelle. En d'autres termes, les classes devraient s'appuyer sur des interfaces plutôt que de dépendre directement d'autres classes. Ce principe, appelé "Dependency Inversion Principle", fait partie des principes de conception SOLID.

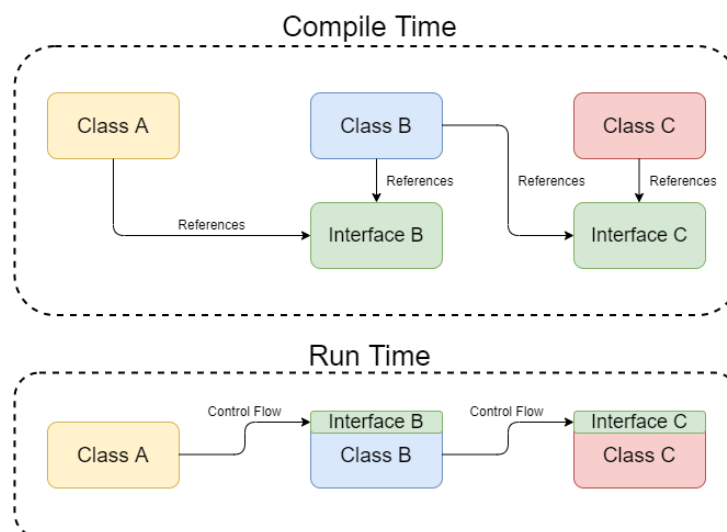


FIGURE 3.2 – Schéma des dépendances de compilation inversé

Source: <https://code-maze.com/wp-content/uploads/2020/08/inverted-flow.png>

Ce principe rend l'application plus flexible et adaptable. En utilisant des interfaces, il devient plus facile de remplacer ou de modifier des composants sans impacter les autres parties du code. Cela améliore également la testabilité en permettant des tests unitaires plus isolés et efficaces. De plus, ce principe facilite l'application de l'injection de dépendances, une pratique clé dans l'architecture logicielle moderne. L. MICROSOFT, 2023b

3.3.2 Application tout-en-un

Lorsqu'on commence à travailler avec les technologies .NET et le framework ASP.NET Core, il est courant de démarrer une application avec un seul projet (un fichier .csproj). Cette méthode est idéale pour les petites applications ou les prototypes. Cependant, au fur et à mesure que l'application grandit en complexité, il devient avantageux de structurer l'application en plusieurs couches distinctes. Souvent, en tant que débutant, les guides de Microsoft nous orientent vers une organisation monolithique.

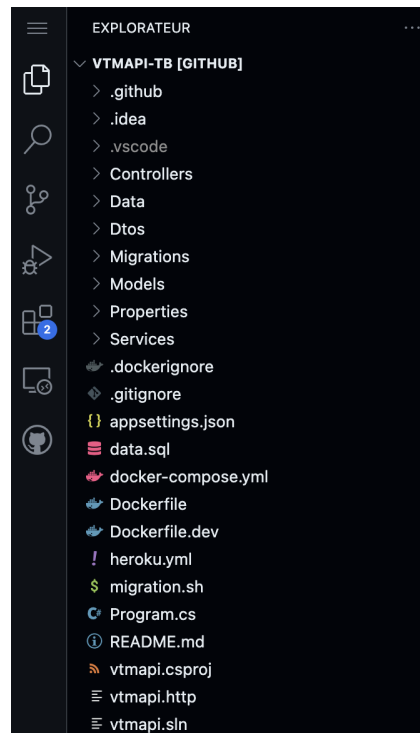


FIGURE 3.3 – Application tout-en-un, organisation de base fournie par Microsoft
Source: de l'auteur

Dans une architecture initiale, toute la logique d'application est contenue dans un seul projet. Ce projet est ensuite compilé en un seul "assembly" (un fichier .dll) et déployé en tant qu'unité unique. Pour organiser la logique de l'application, on utilise des dossiers. Par défaut, on trouve des dossiers séparés selon le schéma MVC, ainsi que pour les services et les données.

Bien que cette approche monolithique soit simple, elle comporte des inconvénients. Le nombre de fichiers et de dossiers augmente avec la taille du projet, ce qui complique la gestion. La logique métier se retrouve dispersée entre les dossiers "Models" et "Services", rendant les dépendances entre les classes difficiles à suivre. Cela peut mener à une mauvaise organisation et à du code enchevêtré, souvent appelé "code spaghetti".

Pour éviter ces problèmes, il est courant de faire évoluer l'application vers une solution à projets multiples, où chaque projet correspond à une couche spécifique de l'application. L. MICROSOFT, 2023a

3.3.3 Architecture N-tier

L'architecture N-tier est une méthode couramment utilisée pour structurer les applications ASP.NET Core, permettant de segmenter l'application en plusieurs couches logiques, chacune ayant une responsabilité spécifique. Typiquement, cette architecture se compose de trois couches principales :

- **Couche UI** : Elle présente les informations à l'utilisateur et gère ses requêtes. Cette couche inclut des composants comme les interfaces utilisateur, les pages web ou les points de terminaison d'API. Elle interagit uniquement avec la couche BLL.
- **Couche BLL** : La couche de logique métier applique les règles métiers, maintient l'intégrité des données et coordonne les interactions entre les différentes parties du système. Elle fait appel à la couche DAL pour l'accès aux données.
- **Couche DAL** : Elle gère les interactions avec le stockage de données, tel qu'une base de données ou des services externes. En utilisant des outils comme Entity Framework, elle mappe les objets de l'application aux tables de la base de données, simplifiant ainsi le développement et améliorant la maintenabilité.

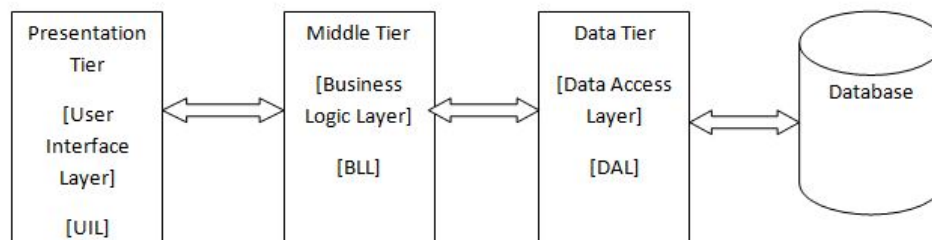


Fig: 3 Tier Applications

FIGURE 3.4 – *Architecture en N couches conventionnelle*

Source: <http://www.sitecorelessons.com/2013/07/how-to-create-3-tier-layer-architecture.html>

Cette organisation garantit que chaque couche a une responsabilité distincte, et que la couche UI ne communique jamais directement avec la couche DAL. De cette manière, la logique métier est encapsulée dans la BLL, tandis que les opérations de persistance des données sont gérées par la DAL.

Bien que l'application soit structurée en plusieurs projets pour une meilleure organisation du code, elle est déployée comme une seule unité. Lors du déploiement, tous les projets sont combinés en un seul package déployable.

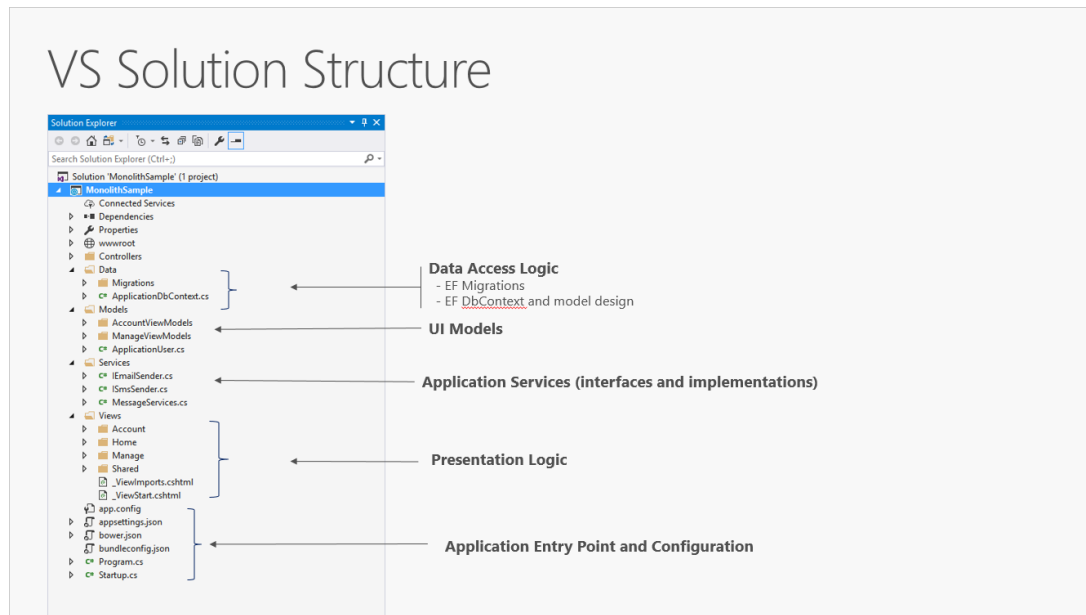


FIGURE 3.5 – Représentation d'un projet ASP.NET implémentant une architecture N-Tier dans Visual Studio

Source: <https://learn.microsoft.com/fr-fr/dotnet/architecture/modern-web-apps-azure/media/image5-1.png>

Dans cette architecture en couches, la compilation s'effectue de haut en bas : la couche UI dépend de la couche BLL, et la couche BLL dépend de la couche DAL, créant ainsi un couplage fort. Cependant, ce couplage peut poser des défis, notamment lorsqu'il s'agit de tester la couche BLL de manière isolée.

Un problème courant de cette structure est la nécessité d'utiliser une base de données de test pour vérifier le bon fonctionnement de la couche BLL. Étant donné que la BLL dépend directement de la DAL pour accéder aux données, tester la BLL nécessite que la DAL fonctionne correctement. Cela implique que les tests de la BLL doivent également interagir avec la base de données réelle, ce qui peut compliquer la mise en place de scénarios de test réalistes.

Une base de données de test permet de simuler ces interactions et de vérifier comment la BLL gère les données dans diverses situations. Cependant, cela ajoute de la complexité au processus de développement, car la mise en place et la maintenance d'une base de données de test exigent des efforts supplémentaires. De plus, les interactions avec la base de données peuvent ralentir les tests unitaires et les rendre plus fragiles.

Pour résoudre ces problèmes, la Clean Architecture propose une approche plus flexible en séparant strictement les responsabilités et en découplant les dépendances entre les couches, offrant ainsi une solution plus modulaire et maintenable. L. MICROSOFT, 2023a

3.3.4 Clean Architecture

La Clean Architecture est une approche de conception de logiciels qui vise à séparer les différentes parties d'une application pour qu'elles soient indépendantes les unes des autres. Ce modèle, qui a été popularisé par Robert C. Martin dans son livre "Clean Architecture : A Craftsman's Guide to Software Structure and Design", est conçu pour rendre les systèmes plus modulaires, évolutifs, et faciles à maintenir.

L'idée principale est d'organiser une application en couches, où le cœur de l'application (la logique métier) est isolé des détails techniques comme l'interface utilisateur ou la base de données. Cela permet de changer ou de remplacer ces détails sans affecter le cœur de l'application.

Plutôt que de faire dépendre le noyau (couches Domaine et Application) des détails d'implémentation (comme l'accès à la base de données), la Clean Architecture inverse ces dépendances. Le noyau définit des interfaces abstraites que les couches externes (comme l'infrastructure) doivent implémenter. Cela garantit que les changements dans les couches externes n'affectent pas le noyau de l'application. L. MICROSOFT, 2023a ; POUCHELET, 2024 ; RULES, 2024

La Clean Architecture est souvent représentée par des cercles concentriques, comme les couches d'un oignon. Voici un aperçu des quatre principales couches : RULES, 2024

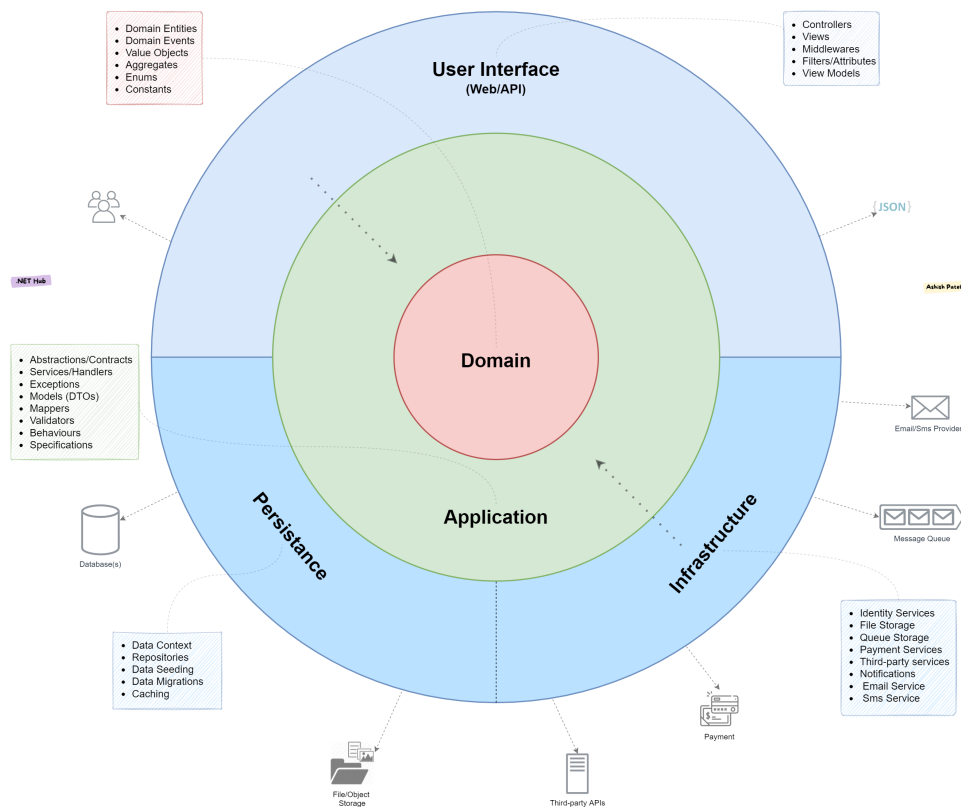


FIGURE 3.6 – Clean Architecture

Source: <https://user-images.githubusercontent.com/93929557/233073112-ab54f354-dca6-4c9e-b98a-9380231377c3.png>

3.3.4.1 Couche Domaine

La couche Domaine contient la logique métier fondamentale. Elle représente le cœur de l'application et ne dépend d'aucune autre couche. Elle définit les modèles et les entités qui décrivent les concepts clés du métier. C'est ici que se trouvent les règles et logiques centrales du système. RULES, 2024

3.3.4.2 Couche Application

La couche Application orchestre le flux de données dans l'application. Elle dépend de la couche Domaine pour effectuer des opérations, mais elle reste indépendante des détails techniques comme l'interface utilisateur ou l'accès aux données. Elle définit les cas d'utilisation de l'application sous forme de services, en s'appuyant sur les abstractions fournies par la couche Domaine.

Par exemple, pour accéder aux données, la couche Application ne dépend pas directement de la base de données. Au lieu de cela, elle définit des interfaces abstraites que la couche Infrastructure implémente. Cela permet de changer facilement la manière dont les données sont stockées ou récupérées sans modifier la logique métier. RULES, 2024

3.3.4.3 Couche Infrastructure

La couche Infrastructure contient les implémentations concrètes des interfaces définies dans la couche Application. Elle gère les interactions avec les systèmes externes comme les bases de données, les services externes, ou les API. Cette couche dépend des abstractions définies par la couche Application, garantissant que les détails techniques n'affectent pas la logique métier. RULES, 2024

3.3.4.4 Couche Présentation

La couche Présentation est responsable de l'interaction avec l'utilisateur. Elle peut être une interface graphique, une API Web, ou même une ligne de commande. Cette couche prend les requêtes des utilisateurs, les transmet à la couche Application, et renvoie les réponses appropriées. Elle ne doit contenir aucune logique métier, se concentrant uniquement sur la présentation des données. RULES, 2024

3.4 Docker

Docker permet d'embarquer une application dans un conteneur virtuel qui peut s'exécuter sur n'importe quelle machine. Il simplifie grandement la gestion des dépendances au sein d'un projet. Cette technologie vise à faciliter le déploiement des applications et la gestion des ressources matérielles et logicielles nécessaires à leur fonctionnement. Docker est en partie proposé en open source (sous licence Apache 2.0) par une société américaine du même nom, fondée par le Français Solomon Hykes. ROUSSEZ, 2017

Docker se distingue par son utilisation de conteneurs et leur orchestration, ce qui le différencie significativement des machines virtuelles. DATASCIENTEST, 2020

3.4.1 Conteneurisation VS Virtualisation

Contrairement aux machines virtuelles, les conteneurs n'incluent pas de système d'exploitation complet, ce qui les rend beaucoup plus légers (de l'ordre de quelques Mo). Les conteneurs n'ont pas besoin d'activer un second système d'exploitation, ce qui se traduit par un démarrage beaucoup plus rapide. De plus, ils peuvent être facilement migrés d'une machine physique à une autre ou d'un cloud à un autre. DATASCIENTEST, 2020 ; ROUSSEZ, 2017.

Containers vs. VMs

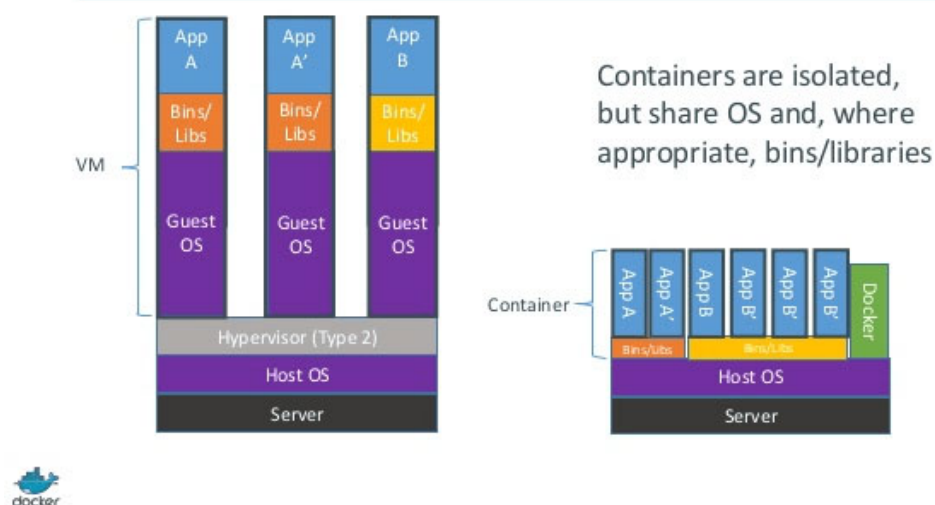


FIGURE 3.7 – Conteneurisation VS Virtualisation

Source: <https://julien.io/content/images/2017/12/containers-versus-virtual-machines-docker-inc-rightscale.jpg>

3.4.2 Docker en production

Grâce à Docker, il est possible de conteneuriser une application en isolant chaque couche et chaque composant dans des conteneurs distincts. Cela correspond au concept de l'architecture de microservices, où chaque conteneur représente une partie spécifique de l'application. Par exemple, une application web peut avoir trois conteneurs : un conteneur pour le frontend, un conteneur pour le backend, et un conteneur pour la base de données. Ces conteneurs sont légers et peuvent s'exécuter sur les machines souhaitées.

Les conteneurs Docker étant portables d'une infrastructure à une autre, il devient possible de dupliquer une application sur plusieurs serveurs (mirroring) et de répartir la charge de travail entre différents clouds (équilibrage de charge). Cela permet également de mettre en place des plans de reprise ou de continuité d'activité entre clouds, voire de transférer un projet d'un fournisseur cloud à un autre.

En outre, Docker présente l'avantage de limiter les mises à jour aux seules parties du conteneur qui en ont besoin. Cela signifie que lorsque des modifications sont nécessaires, il n'est pas nécessaire de reconstruire ou de redéployer l'ensemble de l'application. Au lieu de cela, seules les parties spécifiques du conteneur nécessitant des mises à jour sont modifiées. Ce qui simplifie la gestion des mises à jour. ROUSSEZ, 2017

3.4.3 Exécuter une base de données dans un conteneur

Initialement, Docker n'a pas été conçu pour les services avec état. L'un des principaux arguments de vente des conteneurs est qu'ils peuvent être arrêtés et démarrés à volonté, en s'appuyant généralement sur une source de données externe, comme une base de données, pour stocker leur état. Ces conteneurs sont dits "stateless" car toutes les données générées à l'intérieur sont éphémères et sont détruites lorsque le conteneur est supprimé.

Avec le temps Docker est devenu capable d'exécuter des charges de travail avec état, dites "stateful", nécessite une gestion différente. Les conteneurs stateful sont conçus pour conserver et gérer les données persistantes tout au long de leur cycle de vie. Ils mémorisent des informations telles que les configurations, les données de session ou le contenu des bases de données. Pour ce faire, Docker propose des outils de gestion des volumes qui permettent de monter un emplacement sur la machine hôte vers un emplacement dans le conteneur. Cela assure que les données sont stockées de manière persistante, même lorsque le conteneur est arrêté ou déplacé. HEDDINGS, 2020 ; ROUSSEZ, 2017

3.4.4 Est-ce nécessaire de déployer une base de données conteneurisée ?

En général, la réponse est "pas vraiment". Bien que Docker ait beaucoup évolué depuis sa création et que conteneuriser des bases de données ne soit plus une mauvaise idée, cela reste complexe pour la plupart des charges de travail. Les avantages ne compensent pas toujours les inconvénients. Pour comprendre pourquoi, voici quelques points importants à considérer si Docker est envisagé d'être utilisé pour des bases de données. HEDDINGS, 2020

3.4.4.1 Evolutivité

La mise à l'échelle des bases de données est un défi majeur lorsque la charge de travail varie. L'évolutivité horizontale, qui consiste à ajouter plus de serveurs pour partager la charge, est souvent la meilleure solution. Mais il faut s'assurer que la base de données peut effectivement se développer de cette manière.

La plupart des bases de données peuvent facilement gérer un plus grand nombre de lectures (c'est-à-dire la consultation des données) en ajoutant des serveurs supplémentaires. Cependant, c'est plus compliqué pour les écritures (c'est-à-dire l'insertion, la mise à jour et la suppression des données). Les SGBDR (Systèmes de Gestion de Bases de Données Relationnelles) n'ont généralement pas la capacité de répartir ces opérations d'écriture entre plusieurs serveurs. Cependant, certaines bases de données utilisent le partitionnement et des rôles de répartition de charge pour y parvenir.

Dans un environnement où chaque conteneur fonctionne de manière isolée, il est crucial de vérifier si la base de données peut être mise à l'échelle tout en étant exécutée sous forme de conteneurs. Même avec le partitionnement, les données doivent rester accessibles et pouvoir communiquer avec le nœud master ou les autres nœuds du cluster. Les conteneurs exécutant des bases de données doivent également maintenir cette communication de manière fiable.

Kubernetes (K8s) peut faciliter cette communication entre conteneurs, mais cela rend l'environnement plus complexe. Dans de nombreux cas, il peut être plus simple et efficace d'utiliser des outils de gestion de bases de données traditionnels sans recourir aux conteneurs. GADGE, 2019

3.4.4.2 Facilité d'installation et de configuration

À quelle fréquence est-il réellement nécessaire d'installer une base de données en environnement de production chaque mois ? Chaque environnement de production est unique en raison des différentes charges de travail, ce qui nécessite souvent la création de nouvelles images ou de nouveaux fichiers d'exécution Docker, même si l'image de base reste la même.

Alors, un conteneur est-il vraiment nécessaire dans ce cas ? Si les bases de données sont faciles à installer et configurer, pourquoi ne pas utiliser des scripts d'automatisation comme Ansible pour gérer ces tâches, ainsi que la gestion des bases de données ? Cela pourrait simplifier le processus en évitant d'ajouter la complexité associée à la gestion des conteneurs. GADGE, 2019

3.4.4.3 Portabilité

L'un des principaux avantages des conteneurs est leur portabilité. Il est facile de transférer un conteneur (et ses images) avec son environnement complet vers un autre système. Cependant, la question se pose : à quelle fréquence est-il vraiment nécessaire de déplacer une base de données de production ?

Imaginons le service Veetamine, qui intègre une architecture microservices. Veetamine utilise plusieurs microservices conteneurisés pour traiter les données des machines à café, gérer les utilisateurs et générer la facturation. Ces microservices peuvent facilement être déplacés d'un environnement de développement local à un cloud public, car ils sont encapsulés dans des conteneurs Docker.

En revanche, pour la base de données de Veetamine, même si elle est exécutée dans un conteneur, elle est généralement mappée sur un volume externe pour stocker les données de manière persistante. Cela signifie que les données ne sont pas stockées à l'intérieur du conteneur, mais sur un disque externe. Si le conteneur est déplacé vers un autre environnement, il faut remapper ce volume externe dans le nouvel environnement.

Est-il prudent de prendre ce risque avec des données de production ? Les bases de données conteneurisées ajoutent un niveau de complexité supplémentaire pour la gestion des volumes de données, ce qui peut ne pas justifier les avantages en termes de portabilité pour des bases de données critiques en production. GADGE, 2019

3.4.5 Conclusion

Si l'objectif est de simplifier la gestion des bases de données, Docker n'est pas l'outil idéal. Il introduit une complexité inutile pour une charge de travail qui peut facilement s'exécuter sur un serveur standard. Il est souvent plus avantageux d'utiliser un service de base de données entièrement géré, qui offre une grande partie de l'automatisation de Docker sans les tracas de gestion.

En revanche, Docker peut être particulièrement utile pour les environnements de développement. Il permet de créer facilement de nouvelles bases de données avec des configurations variées, facilitant ainsi les tests rapides. Cependant, en production, les exigences sont généralement plus strictes et les bénéfices de Docker sont moins évidents. HEDDINGS, 2020

4 | Choix préalables au développement

4.1 Contrôleurs ou API Minimales

Dans le cadre de mon travail de Bachelor, j'ai choisi de privilégier l'utilisation des contrôleurs pour les raisons suivantes :

- **Organisation et clarté du code** : Après avoir effectué quelques tests, j'ai constaté que l'utilisation des contrôleurs permet une organisation du code plus claire et structurée. L'implémentation des points de terminaison directement dans le fichier "Program.cs" ou dans un fichier séparé peut rendre la lecture du code plus complexe. Bien que mon expérience avec les contrôleurs influence ce choix, cette approche me permet de gagner du temps et d'améliorer l'efficacité du développement de l'API.
- **Homogénéité avec le projet existant** : Actuellement, le projet Veetamine utilise également des contrôleurs. En continuant avec cette méthode, je maintiens la cohérence du projet. Cela facilitera la transition entre les différentes parties du projet et améliorera la collaboration avec les autres développeurs, déjà familiers avec cette approche.
- **Adapté à mon projet** : D'après mes recherches dans le chapitre de l'état de l'art, les API minimalistes sont particulièrement adaptées pour des projets de type prototype, ce qui n'est pas le cas de mon projet. De plus, étant donné que je vais intégrer une architecture N-Tier ou la Clean Architecture, l'utilisation de contrôleurs me semble plus appropriée. KUMAR, 2023

4.2 L'architecture

Afin de faire un choix approprié pour mon projet, voici une comparaison entre l'architecture N-Tier et la Clean Architecture.

4.2.1 Comparaison entre l'architecture N-Tier et la Clean Architecture

4.2.1.1 Principes de base

N-Tier : Elle se repose sur la séparation des préoccupations en couches distinctes. Chaque couche a une responsabilité spécifique, généralement divisée en trois couches principales. Le but est de structurer l'application de manière à ce que les couches soient indépendantes les unes des autres, en facilitant ainsi l'encapsulation et la modularité.

Clean Architecture : Elle met l'accent sur l'indépendance des couches internes par rapport aux couches externes. Elle suit les principes SOLID, en particulier le principe d'inversion de dépendance, pour garantir que les détails d'implémentation ne dépendent pas de la logique métier, mais l'inverse. L'objectif est de créer un système modulaire, maintenable et facilement testable. OLESON, 2023

4.2.1.2 Structure

N-Tier : Sa structure est généralement linéaire, avec des couches empilées les unes sur les autres. Chaque couche ne communique qu'avec la couche immédiatement adjacente. Cette architecture est souvent utilisée pour des applications CRUD simples où la logique métier n'est pas très complexe.

Clean Architecture : Elle est organisée en couches concentriques. Au centre, la logique métier est isolée des couches externes (comme l'infrastructure ou l'interface utilisateur). Les couches externes peuvent changer sans affecter la logique centrale, ce qui rend le système plus résilient aux changements. OLESON, 2023

4.2.1.3 Objectifs

N-Tier : Son objectif est d'améliorer la séparation des préoccupations, de faciliter la réutilisation des composants, et de renforcer la sécurité en isolant les différentes couches. Elle est également conçue pour améliorer les performances en répartissant les responsabilités entre différentes couches.

Clean Architecture : Son objectif est de maximiser la flexibilité et la maintenabilité du code. Elle vise à rendre le système adaptable aux changements des exigences en minimisant l'impact de ces changements sur le reste du système. La réutilisation du code et la facilité de test sont également des priorités. OLESON, 2023

4.2.1.4 Complexité de la logique métier

N-Tier : Pour des scénarios où la logique métier est simple ou lorsque l'application est principalement centrée sur des opérations CRUD, l'architecture N-Tier est souvent suffisante et appropriée. Elle permet une implémentation rapide et est plus facile à comprendre pour des équipes ayant moins d'expérience en architectures complexes.

Clean Architecture : Lorsque la logique métier est complexe, critique, et au cœur de l'application, la Clean Architecture est préférée. Elle permet de protéger cette logique des problèmes d'infrastructure et d'assurer que les changements dans les couches externes n'affectent pas le noyau de l'application. OLESON, 2023

4.2.1.5 Conclusion

En conclusion, le choix entre l'architecture N-Tier et la Clean Architecture dépend de la complexité de la logique métier et des besoins spécifiques du projet. Pour des projets simples avec des exigences CRUD, l'architecture N-Tier peut suffire. En revanche, pour des projets complexes où la logique métier est au cœur de l'application, la Clean Architecture offre une meilleure modularité, testabilité, et flexibilité, assurant ainsi une meilleure gestion des changements futurs. OLESON, 2023

4.2.2 Choix de l'architecture

En absolu, l'architecture N-Tier serait appropriée pour mon projet si l'on se concentre uniquement sur les exigences de base. En effet, je ne pense pas que mon projet soit d'une très grande complexité, et l'architecture N-Tier pourrait suffire.

Cependant, je suis conscient de la complexité du projet Veetamine et de l'importance du choix architectural. Dans ce contexte, la Clean Architecture me semble plus adaptée. En tant que développeur junior, il m'est difficile de certifier un choix définitif, mais je perçois cette situation comme une opportunité d'explorer une architecture qui pourrait être très bénéfique pour l'avenir de Veetamine.

Voici les principaux points qui m'ont conduit à choisir la Clean Architecture, au-delà de test :

- **Indépendance des couches** : Cellsmaniak utilise actuellement Invoice Ninja pour la facturation. En choisissant la Clean Architecture, je garantis que les différentes couches de l'application restent indépendantes les unes des autres. Cela signifie que si Cellsmaniak décide de changer d'outil de facturation à l'avenir, l'impact sur le reste de l'application sera minimisé, rendant les ajustements nécessaires plus simples et moins coûteux.
- **Adhérence aux principes SOLID** : Lors de mes recherches sur les meilleures pratiques en ingénierie logicielle, les principes SOLID se sont avérés essentiels. En choisissant la Clean Architecture, qui applique ces principes de manière rigoureuse, je m'assure de les mettre en œuvre efficacement.
- **L'intérêt marqué de Microsoft** : Mes recherches ont révélé un intérêt prononcé de la part de Microsoft pour la Clean Architecture. Cela m'a influencé dans ma réflexion et m'a orienté vers cette approche.

4.3 La base de données

Actuellement, pour le projet Veetamine, la base de données utilisée est SQL Server. Cependant, Cellsmaniak me donne la possibilité d'intégrer n'importe quelle autre base de données relationnelle si cela s'avère approprié.

4.3.1 Mes prérequis

Avant de commencer mes recherches, j'ai établi une liste de critères pour m'assurer que la base de données choisie s'intégrera bien avec mon projet. Voici les critères que j'ai définis :

- **Bonne intégration avec Entity Framework et .NET** : Entity Framework est un ORM (Object-Relational Mapping) qui permet aux développeurs de manipuler des bases de données en utilisant des objets .NET, sans avoir à écrire de code SQL complexe. Cette approche simplifie considérablement l'accès aux données et accélère le développement en réduisant la quantité de code nécessaire pour les opérations courantes. L. MICROSOFT, 2023c
- **Possibilité d'être déployable sur Google Cloud Platform** : Étant donné que Veetamine sera déployé sur Google Cloud Platform dans le futur, il est essentiel que la base de données soit compatible avec cet environnement.
- **Performance en production** : Ce critère vise à écarter les bases de données qui ne sont pas adaptées aux applications en production, comme SQLite. La base de données doit être capable de gérer efficacement des charges de travail élevées, avec une haute disponibilité et une bonne performance sous des charges lourdes.

En tenant compte de ces critères, j'ai identifié trois bases de données qui pourraient convenir à mon projet : SQL Server, PostgreSQL et MySQL. Cependant, il est crucial de reconnaître qu'aucune base de données ne peut satisfaire toutes les exigences d'un projet.

4.3.2 Introduction : MySQL et PostgreSQL

MySQL est un système de gestion de base de données relationnelle (SGBDR) qui permet de stocker des données sous forme de tableaux avec des lignes et des colonnes. C'est un système populaire qui alimente de nombreuses applications, des sites web dynamiques aux systèmes intégrés. MySQL est disponible sous une licence open source qui permet d'utiliser et de modifier librement le code source.

PostgreSQL est un système de gestion de base de données objet-relationnelle en open source, publié sous la licence PostgreSQL. Il est compatible avec les bases de données relationnelles (SQL) et non relationnelles (JSON) et offre des fonctions SQL plus avancées que celles de MySQL. Il propose une plus grande flexibilité au niveau des types de données, une meilleure capacité de mise à l'échelle, une gestion avancée de la simultanéité et une intégrité des données renforcée. AWS, 2024

4.3.3 Comment choisir entre MySQL et PostgreSQL ?

PostgreSQL

- Mieux adapté aux applications d'entreprise nécessitant des opérations d'écriture fréquentes et des requêtes complexes.
- Utilise le contrôle de simultanéité multiversion (MVCC) sans verrous de lecture-écriture, offrant de meilleures performances pour les écritures fréquentes et simultanées.
- Nécessite une configuration plus complexe et consomme davantage de mémoire pour gérer plusieurs utilisateurs simultanément.

MySQL

- Plus adapté aux débutants avec une courbe d'apprentissage plus courte et une configuration plus simple.
- Utilise des verrous d'écriture pour la simultanéité, ce qui peut entraîner des temps d'attente pour les opérations d'écriture concurrentes.
- Idéal pour les applications nécessitant des lectures de données fréquentes.

PostgreSQL semble être un choix intéressant pour ce projet car il offre une plus grande flexibilité en termes de types de données et de capacité de mise à l'échelle. Étant donné que le modèle économique de Veetamine repose entièrement sur les données, il est crucial que la base de données ne soit pas un frein à la croissance de l'entreprise. PostgreSQL pourrait également être un bon choix, car il est particulièrement adapté aux applications d'entreprise nécessitant des opérations d'écriture fréquentes et des requêtes complexes. Bien que PostgreSQL consomme davantage de mémoire pour gérer plusieurs utilisateurs simultanément, cela ne pose pas un problème majeur pour Veetamine, car il ne s'agit pas d'un réseau social nécessitant la gestion d'un grand nombre d'utilisateurs en même temps. AWS, 2024

4.3.4 Microsoft SQL Server vs PostgreSQL

Je souhaite maintenant comparer PostgreSQL avec Microsoft SQL Server pour déterminer laquelle de ces deux bases de données est la plus appropriée pour mon projet.

Selon Google Cloud, les options SQL les plus courantes sont PostgreSQL et SQL Server. Bien que ces deux systèmes partagent de nombreuses fonctionnalités essentielles, il existe quelques différences importantes. La principale étant que PostgreSQL est open source, tandis que SQL Server est un produit commercial sous licence de Microsoft CLOUD, 2024.

4.3.4.1 Introduction : Microsoft SQL

Microsoft SQL Server permet de gérer et de stocker des données pour répondre à divers cas d'utilisation en entreprise, notamment pour l'informatique décisionnelle (Business Intelligence), le traitement des transactions, l'analyse de données et les services de machine learning.

SQL Server utilise une structure de table basée sur les lignes, ce qui permet de connecter des éléments de données associés provenant de différentes tables sans avoir à stocker les mêmes données plusieurs fois dans une base de données.

Microsoft SQL Server est reconnu pour sa haute disponibilité, ses performances élevées lors du traitement de charges de travail importantes et son intégration facile avec d'autres applications, favorisant ainsi l'informatique décisionnelle à travers l'ensemble de votre infrastructure de données. CLOUD, 2024

4.3.4.2 Tarification

SQL Server, en tant que produit de Microsoft, est disponible sous licence commerciale avec des options d'édition Standard ou Enterprise. Le coût varie entre 3 500\$ et 14 000\$, en fonction de l'édition choisie et des composants nécessaires. Il existe également deux versions gratuites : une édition complète pour les développeurs travaillant sur des charges de travail hors production et une édition Express avec des fonctionnalités et des tailles de base de données limitées.

PostgreSQL est une solution Open Source disponible sous licence PostgreSQL. Ce produit est donc gratuit, quelle que soit son utilisation, y compris à des fins commerciales. D'après le groupe de développement mondial de PostgreSQL, il va rester gratuit et Open Source pour une durée illimitée, et il n'est pas prévu de modifier la licence ni de publier le produit sous une autre licence. CLOUD, 2024

4.3.4.3 Plateformes compatibles

En tant que plateforme open source, PostgreSQL est compatible avec la plupart des principaux systèmes d'exploitation. Il peut être hébergé sur une large gamme de systèmes, notamment Linux, macOS, Windows, BSD et Solaris. De plus, il peut être déployé sur des conteneurs Docker ou sur des clusters Kubernetes.

En revanche, SQL Server a une compatibilité plus limitée en termes de systèmes d'exploitation. Il fonctionne principalement sur Microsoft Windows, Microsoft Server, et dans une certaine mesure sur Linux. CLOUD, 2024

4.3.4.4 Syntaxe et langage

En ce qui concerne la prise en charge des langages de programmation, il existe une grande différence entre SQL Server et PostgreSQL. PostgreSQL est compatible avec une large variété de langages, notamment Python, PHP, Perl, Tcl, .NET, C, C++, Delphi, Java, JavaScript (Node.js), et bien d'autres.

SQL Server, bien que compatible avec plusieurs langages populaires, a une portée légèrement plus restreinte. Il prend en charge Java, JavaScript (Node.js), C#, C++, PHP, Python et Ruby. CLOUD, 2024

4.3.5 Choix de la base de données

Étant donné que Cellsmaniak me donne l'opportunité de changer de base de données, il aurait été dommage de ne pas profiter pour expérimenter une nouvelle base de données. Après avoir étudié les options, j'ai réalisé que PostgreSQL est un excellent choix, au-delà de l'intérêt d'acquérir une nouvelle expérience.

PostgreSQL se distingue par ses fonctionnalités et ses performances, qui sont très proches de celles de SQL Server. Cependant, PostgreSQL, étant open source et donc gratuit, devient une offre très attractive pour une start-up. De plus, PostgreSQL est compatible avec une large gamme de systèmes d'exploitation, ce qui en fait une solution plus flexible que SQL Server.

Pour la réalisation de ce projet, j'ai donc choisi d'utiliser PostgreSQL comme base de données.

4.4 Service Cloud

Cellsmaniak m'a demandé de trouver une alternative moins coûteuse et moins complexe par rapport à Azure et Google Cloud Platform. Le critère principal était de trouver un service cloud capable d'héberger une application .NET basée sur une image Docker ASP.NET Core, avec une base de données locale identique à celle de l'environnement de production.

L'objectif est de minimiser le temps consacré à la configuration d'un environnement de production en trouvant une solution rapide et efficace, car la valeur ajoutée de ce projet réside dans l'automatisation de la facturation. Je me suis donc orienté vers une solution PaaS (Platform as a Service) qui permettrait de déployer l'application sans avoir à gérer l'infrastructure sous-jacente.

4.4.1 Qu'est-ce que le PaaS ?

Le PaaS (Platform as a Service) est une offre de cloud computing où un fournisseur de services met à disposition une plateforme complète pour ses clients. Cette solution permet de fournir des ressources informatiques de manière évolutive, rapide et efficace, tout en réduisant les coûts opérationnels et en augmentant l'agilité des entreprises. Le PaaS est particulièrement avantageux pour les entreprises qui doivent déployer rapidement des applications dans des environnements de développement, de test ou de production, en leur offrant une infrastructure préconfigurée et facile à gérer. ORACLE, 2024

4.4.2 Utilisation de ChatGPT 4o pour obtenir des recommandations

Je me suis tourné vers ChatGPT 4o pour obtenir des recommandations sur les services cloud qui pourraient convenir à mon projet. Voici le prompt que j'ai utilisé :

(Question) : Donne-moi une liste de fournisseurs cloud PaaS bon marché. Le service
 → doit permettre de déployer des applications dockerisées

(Recommandations simplifié) : Heroku, DigitalOcean App Platform, Google Cloud Run,
↔ Render

4.4.3 Choix du service cloud

Après avoir examiné ces recommandations, voici comment j'ai procédé pour choisir le service Heroku :

1. **Tarification abordable** : Étant donné que c'est un environnement de production provisoire pour le projet, il était important de limiter les coûts. Heroku et Render sont les deux services qui offrent les meilleures tarifications. Render offre 30 jours de gratuité, puis passe à un tarif de 7\$ par mois. Heroku propose également une tarification minimum de 7\$ par mois, mais selon l'utilisation, le prix peut augmenter plus sévèrement. Les autres services se situent aux alentours de 15\$ par mois. HEROKU, 2024 ; RENDER, 2024
2. **Compatibilité avec les choix technologiques** : Concernant les choix technologiques pour la réalisation de ce projet, Render et Heroku proposent les mêmes services. C'est-à-dire que le déploiement d'une application dockerisée est possible, l'utilisation d'une base de données PostgreSQL également, et des services de CI/CD peuvent être intégrés, comme GitHub Actions.
3. **Expérience d'un collaborateur** : Selon certaines sources sur internet, comme G2, 2024, il n'est pas évident de les départager, bien qu'une légère tendance en faveur de Render ait été identifiée. Face à cette situation, j'ai décidé de demander l'avis d'un collaborateur. Il s'est avéré qu'il avait déjà utilisé Heroku et qu'il en avait été satisfait. C'est l'expérience de mon collègue qui a fait pencher la balance en faveur de Heroku.

Le choix du service cloud n'est pas d'une importance capitale pour ce projet. Je devais simplement m'assurer que le service choisi répondrait aux besoins de mon projet et ne constituerait pas un obstacle à la réalisation de mon travail.

4.5 Conteneuriser la base de données en production

En me basant sur la conclusion de mon chapitre sur l'état de l'art concernant la conteneurisation des bases de données en production, il apparaît que Docker n'est pas l'outil idéal pour la gestion des bases de données, notamment en raison de la complexité liée à la gestion des volumes de données. En effet, les avantages de la conteneurisation ne compensent toujours pas les inconvénients,

Pour l'environnement de production, j'ai donc décidé d'utiliser le service de base de données géré par Heroku, nommé "Heroku Postgres".

Cependant, pour mon infrastructure de développement, je vais conteneuriser ma base de données PostgreSQL avec Docker. Cela permettra de rendre mon environnement de développement portable et facilement reproductible, répondant ainsi à un besoin identifié au début de ce projet.

De plus, en utilisant Entity Framework, je m'assure que le schéma de ma base de données reste identique entre les environnements de développement et de production. En effet, Entity Framework permet de générer automatiquement le schéma de la base de données à partir du modèle de données de l'application, garantissant ainsi la cohérence des données entre les différents environnements. L. MICROSOFT, 2023d

5 | Méthodologies

5.1 Gestion de projet

5.1.1 Modèle et choix de gestion de projet

Pour la planification de mon travail, j'ai décidé d'adopter une gestion de projet agile en utilisant le framework SCRUM. J'ai adapté ce framework à mon contexte de travail, en tenant compte du fait que je suis seul sur ce projet. Ainsi, des éléments comme les daily scrums, les sprint reviews et les sprint planning ne seront pas pertinents dans ce cadre.

En revanche, SCRUM me sera utile pour obtenir une vision claire du produit grâce au Product Backlog. À partir de ce Product Backlog, je pourrai organiser mes sprints. Ceux-ci seront construits selon une priorisation basée sur l'effort requis pour chaque user story. Pour centraliser ma gestion de projet, j'utiliserai Azure DevOps, un logiciel développé par Microsoft.

Bien que je sois le seul développeur sur ce projet, je suis toujours en communication avec le reste de l'équipe. Le CTO de Veetamine jouera le rôle de Product Owner, tandis que le développeur avec qui j'ai l'habitude de travailler occupera un rôle de référent technique, et non de Scrum Master.

Initialement, je souhaitais utiliser Jira, qui est le logiciel utilisé au sein de Cellsmaniak, mais j'ai rencontré plusieurs problèmes. Le rôle qui m'était attribué ne me permettait pas de gérer certains paramètres. Par exemple, je souhaitais ajouter des "Epics" pour regrouper mes user stories afin de gagner en clarté par rapport aux autres user stories de l'entreprise. De plus, le tableau de bord, censé regrouper les user stories selon leur statut et par sprint, affichait des tickets au lieu des user stories. J'ai estimé que l'utilisation de Jira nécessiterait trop de temps pour configurer ma gestion de projet. Par conséquent, j'ai choisi d'utiliser un logiciel avec lequel j'ai déjà de l'expérience.

5.1.2 Planification du projet

J'ai planifié quatre sprints (de 0 à 3) de deux semaines chacun. Le sprint 0 n'est pas intégré à ma gestion de projet, mais son objectif est de préparer un environnement de travail prêt à démarrer pour le sprint 1. Cela inclut la lecture de documentations des technologies que je vais utiliser, la réalisation de quelques mini-projets tests et tutoriels pour me familiariser avec .NET, par exemple. L'analyse de l'existant a également été réalisée en parallèle du sprint 0. Volontairement, je me réserve une dernière semaine après le sprint 3 pour effectuer des tâches de nettoyage.

5.1.2.1 Sprint 1

Nom de la user story : En tant que développeur, je veux que mon projet suive un flux de travail d'industrialisation incluant l'intégration continue (CI) et le déploiement continu (CD) afin de garantir la qualité et l'efficacité des déploiements.

L'objectif de ce premier sprint est de réaliser le plus tôt possible une itération complète du cycle de développement afin d'éviter les surprises en fin de projet. Cela me permettra de m'assurer que mon environnement de développement et de production sont opérationnels.

Les défis à relever sont les suivants :

- Comment établir la communication entre mes deux applications conteneurisées (ASP.NET et Invoice Ninja) ?
- Quel service cloud utiliser pour déployer mon application ?
- Quelle méthode utiliser pour gérer les variables d'environnement en développement et en production ?
- Comment configurer un pipeline CI/CD selon l'environnement de production ?

Puisqu'il s'agit de la première user story, elle est classée avec la priorité la plus élevée (4) et a été estimée à un effort de 5, conformément à la suite de Fibonacci.

5.1.2.2 Sprint 2

Nom des user stories :

- En tant qu'administrateur VTM, je veux que la facture de l'opérateur soit automatiquement transmise à Ninja CRM via l'API.
- En tant qu'administrateur VTM, je souhaite que les factures soient générées automatiquement au début de chaque mois.

L'objectif de ce sprint est d'intégrer la logique métier de l'application et de mettre en place une tâche automatisée. Ce sprint constitue le cœur de mon projet.

Compte tenu de l'importance de la logique métier, ces user stories sont classées avec une priorité de 3. L'effort requis pour la première user story est estimé à 8, tandis que la seconde est évaluée à 5.

Les défis à relever sont les suivants :

- Face à un délai relativement court, comment prioriser les fonctionnalités essentielles sans compromettre l'avancement du projet ?
- Comment mapper les données entre mon API et Ninja Invoice ?
- Comment garantir que toutes les factures sont correctement envoyées au bon nombre de clients ?

- Comment mettre en œuvre une tâche planifiée pour générer les factures de manière fiable ?

5.1.2.3 Sprint 3

Nom des user stories :

- En tant que développeur, je veux écrire et exécuter des tests unitaires pour la fonctionnalité de génération de factures afin de m'assurer qu'elle fonctionne correctement et sans erreurs.
- En tant que développeur, je veux ajuster mon pipeline CI/CD afin d'assurer une intégration fluide et sans erreur dans l'environnement de production.

L'objectif de ce sprint est de renforcer la qualité du code en implémentant des tests unitaires et en optimisant le pipeline de déploiement.

Le principal défi de ce sprint réside dans la maîtrise des outils de test de .NET. Ces user stories ont une priorité de 2, avec un effort estimé à 3. Bien que ces tests soient essentiels pour la stabilité de l'application, ils nécessitent une attention particulière pour garantir une couverture adéquate et une intégration réussie.

5.2 Outils et technologies

5.2.1 Environnement de développement

Auparavant, j'utilisais Visual Studio Code sur ma machine macOS. Je l'ai remplacé par **Rider de JetBrains**, car selon Jon Mickel Inza publié dans le magasin "Programmez !", c'est un IDE performant et une excellente alternative à Visual Studio pour les développeurs utilisant macOS. Ce choix m'a permis d'améliorer mon utilisation de .NET. Ce changement a été facilité par le fait que la HES-SO de Sierre nous fournit une licence. Je ne regrette en rien ce choix, car beaucoup de documentations ou de tutoriels utilisent Visual Studio qui est très proche de Rider en terme d'expérience utilisateur, ce qui m'a permis de suivre les tutoriels sans problème par conséquent un meilleur apprentissage. De plus, Rider intègre une gestion de package NuGet qui rend l'installation de dépendances très simple. Et selon mon expérience, Rider identifie les erreurs de manière plus précise que Visual Studio Code. INZA, 2024

Docker Desktop est un outil que j'avais déjà installé sur ma machine. Je l'ai principalement utilisé comme support visuel, ce qui me permettait d'identifier rapidement si mes conteneurs étaient en cours d'exécution. Dans mon cas, Docker Desktop n'était pas indispensable car les commandes Docker CLI répondaient à mes besoins. J'aurais pu utiliser des alternatives comme Portainer, mais je n'ai pas jugé nécessaire de l'installer.

Postman, déjà installé sur ma machine, s'est avéré être un outil inestimable pour tester mon API ainsi que celle d'Invoice Ninja, que ce soit en environnement local ou en production. Grâce à Postman, j'ai pu facilement vérifier les réponses de l'API d'Invoice Ninja et identifier les données retournées en cas de succès. L'interface graphique intuitive de Postman m'a permis de créer et de tester rapidement des requêtes HTTP, ce qui m'a fait gagner un temps précieux. Plutôt que d'écrire du code en C# pour chaque requête, j'ai pu simuler et valider les appels API directement dans Postman, ce qui a considérablement simplifié le processus de développement.

5.2.2 Outils de productivités

En suivant la méthode d'apprentissage de Team Academy, nous sommes souvent amenés à nous auto-évaluer pour identifier nos forces et nos faiblesses. En les reconnaissant, nous pouvons déterminer des axes d'amélioration et, sur la base de nos points forts, trouver des méthodes d'apprentissage adaptées.

Me reconnaissant principalement comme une personne visuelle, j'ai choisi de maximiser l'utilisation de schémas pour approfondir ma compréhension du projet. La création de schémas m'a non seulement aidé à clarifier les concepts pour moi-même, mais elle a également facilité des discussions plus constructives avec les autres membres de l'équipe.

Pour ce faire, j'ai utilisé Excalidraw, une application en ligne collaborative, simple d'utilisation et sans inscription préalable.

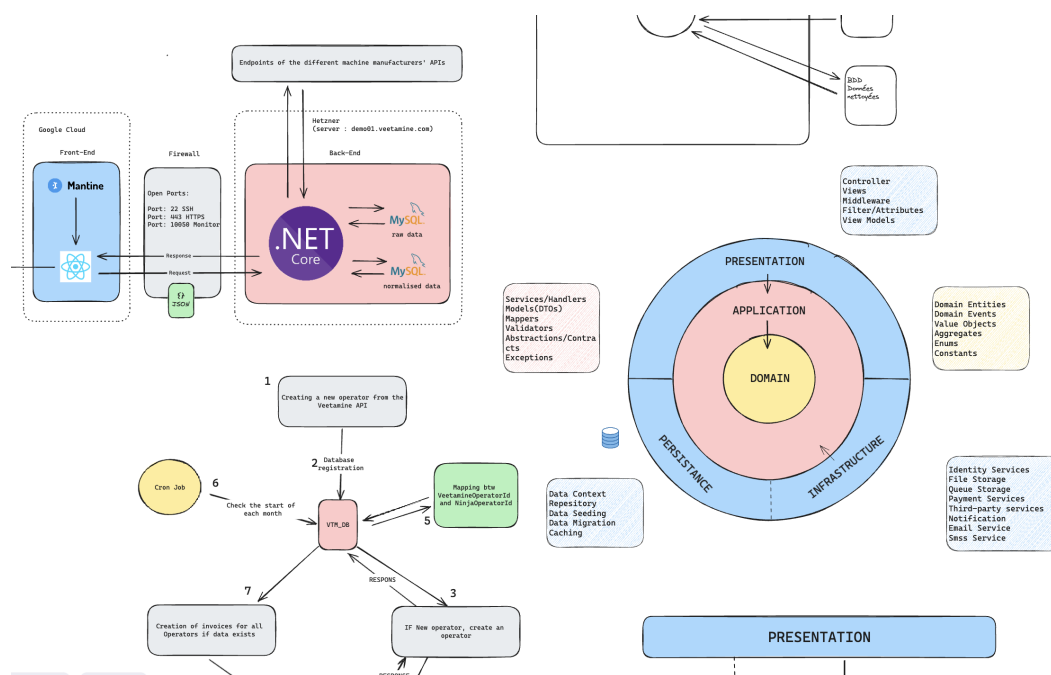


FIGURE 5.1 – Tableau blanc numérique Excalidraw
Source: de l'auteur

L'un des défis majeurs auxquels j'ai été confronté est l'absence de documentation technique détaillée pour le projet. Les documents existants étaient principalement destinés à des fins commerciales. Pour combler ce manque, j'ai dû m'appuyer sur des discussions, des recherches approfondies dans les dépôts GitHub, et poser des questions ciblées. Grâce à cette méthode de schématisation, j'ai pu recréer une représentation claire et visuelle du projet existant, ce qui s'est révélé être une approche d'acquisition de connaissances particulièrement efficace pour moi.

Notion a été un outil indispensable tout au long de ma formation, et il joue aujourd'hui un rôle crucial dans la prise de notes et la documentation. Par exemple, je n'avais pas beaucoup d'expérience avec les commandes Docker. Pour éviter de chercher des commandes sur Internet à chaque fois, bien que l'intelligence artificielle puisse nous faire gagner du temps sur ce type de requête, j'ai créé un onglet dédié à Docker dans Notion. Cela m'a permis de gagner du temps et d'apprendre également les commandes.

5.2.3 Outils de versionnage

Git est de loin le système de contrôle de version distribué le plus utilisé aujourd'hui, et Cellsmaniak ne fait pas exception. Cellsmaniak utilise également GitHub, un service cloud qui fournit une gamme complète de fonctionnalités pour le développement logiciel et qui repose sur Git pour le contrôle de version. Bien que des alternatives comme GitLab ou Bitbucket offrent des fonctionnalités similaires, j'ai décidé d'utiliser GitHub pour des raisons de conformité et par habitude avec l'entreprise. ATLASSIAN, 2024

Étapes de mise en œuvre :

1. **Initialisation du dépôt Git** : J'ai commencé par initialiser le dépôt Git localement en utilisant la commande "git init", puis j'ai configuré les branches principales en fonction du workflow défini (voir section 5.2.3.1).
2. **Utilisation des commandes Git CLI** : Bien que les IDEs offrent une intégration fluide avec Git, je préfère utiliser les commandes CLI. Par exemple, pour créer une nouvelle branche, j'utilise "git checkout -b feat/xxx".
3. **Mise en place de GitHub Actions pour CI/CD** : J'ai configuré un workflow GitHub Actions en définissant un fichier YAML à la racine de mon projet. Ce workflow automatise les processus CI/CD, déclenchés par un commit ou une pull request sur la branche main.
4. **Gestion des secrets avec GitHub** : Les variables d'environnement sensibles, telles que les clés API, sont gérées via GitHub Secrets pour garantir la sécurité et la conformité. Cela permet leur utilisation directe dans les workflows CI/CD sans les exposer dans le code source.

GitHub offre également d'autres fonctionnalités telles que la gestion des issues (tickets de suivi de bugs et de fonctionnalités), les wikis pour la documentation, et l'hébergement de sites web statiques via GitHub Pages. Cependant, pour ce projet, je n'ai pas eu besoin d'utiliser ces fonctions WIKIPEDIA, 2024a.

5.2.3.1 Workflow de branches

Une des principales fonctionnalités de Git est l'utilisation de branches pour isoler les modifications et les fonctionnalités en cours de développement. Cependant, il est important de définir un workflow de branches pour garantir une collaboration efficace et éviter les conflits.

Bien que le besoin de garantir une collaboration puisse sembler moins crucial dans mon cas, étant donné que je travaille seul sur ce projet, je souhaitais tout de même adopter une structure qui assure une version stable de mon application en tout temps. Il existe des modèles de workflow de branches populaires, comme Gitflow, GitHub Flow et GitLab Flow PORTO, 2018. J'ai choisi de définir mon propre workflow en utilisant les branches suivantes :

- **main** : La branche principale de mon projet, qui contient le code de production. Les commits sur cette branche déclenchent un déploiement automatique sur mon service cloud (Heroku).
- **feat/xxx** : Les branches de fonctionnalités, qui contiennent les modifications pour une fonctionnalité spécifique. Une fois la fonctionnalité terminée, je fusionne cette branche avec la branche **main**.

Je dois dire que je n'ai jamais rencontré de gros conflits de fusion, probablement parce que je fusionnais mes branches régulièrement. Cependant, il est arrivé que la branche de production soit cassée et j'ai dû effectuer des corrections directement sur celle-ci. Une branche de développement aurait été utile, liée à un environnement de développement, pour tester les nouvelles fonctionnalités avant de les déployer sur la branche principale. Toutefois, j'ai souhaité éviter des frais supplémentaires.

5.3 Processus de développement

Cette section décrit les étapes suivies pour concevoir, développer, tester, et déployer l'application dans un environnement de production. Chaque étape a été guidée par les sprints que j'ai planifiés ainsi que par les user stories associées, assurant ainsi une logique de développement.

5.3.1 Étape 1 : Analyse et Planification

La première étape consiste à redéfinir le processus de génération de facture en fonction de l'existant.

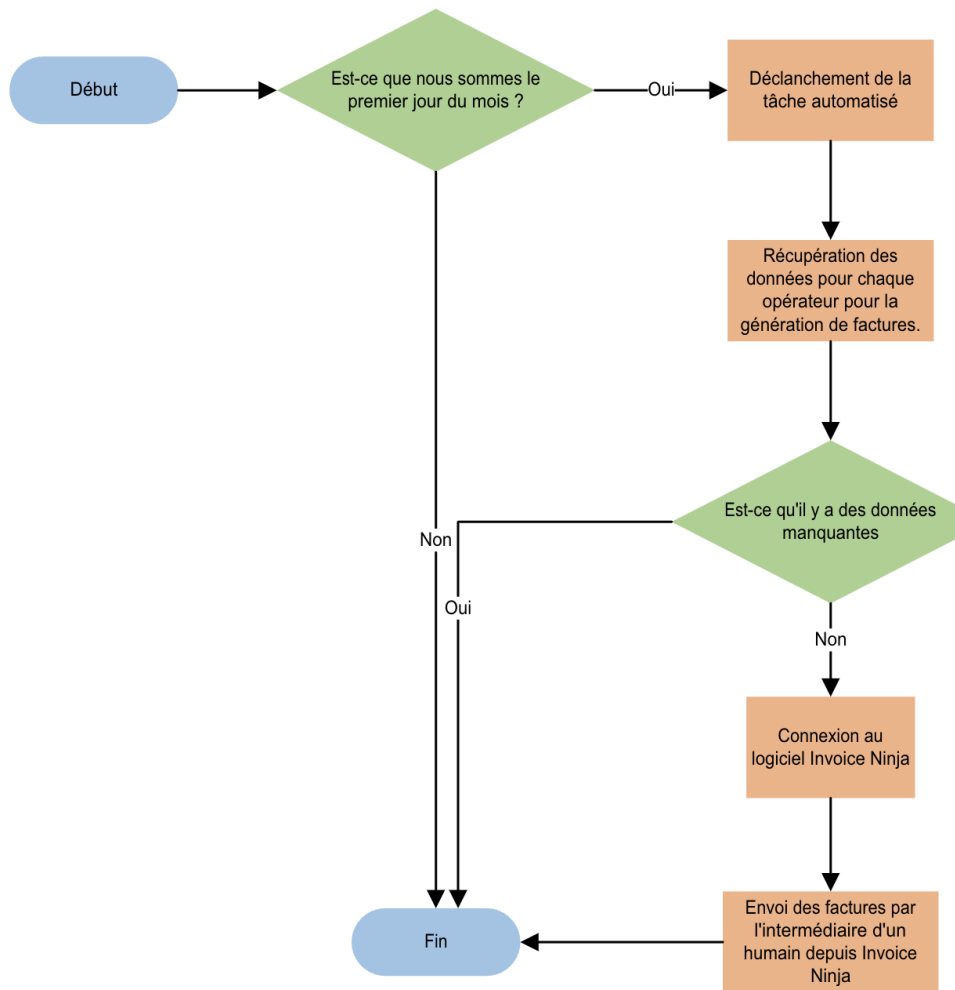


FIGURE 5.2 – *Processus de génération de factures*
Source: de l'auteur

5.3.1.1 Identification des limites du processus actuel

Cette version 1 est une première ébauche et ne gère pas encore tous les cas possibles, notamment en ce qui concerne l'absence de certaines données de facturation. Actuellement, si des données essentielles manquent, le système affichera un message d'erreur dans la console, empêchant ainsi la génération de la facture. Cependant, l'application continuera de fonctionner sans interruption.

Toutefois, il serait intéressant de notifier l'utilisateur dans les deux cas, que ce soit en cas d'échec ou de réussite. J'ai identifié plusieurs solutions potentielles. Par exemple, il existe une série d'endpoints dans l'API de Veetamine dédiés aux notifications, ou l'implémentation d'un service de mailing. Néanmoins, j'ai jugé que ce cas serait à traiter dans une prochaine itération, mon objectif principal étant de générer les factures.

5.3.2 Étape 2 : Mise en place de l'environnement de développement

Pour débuter, j'ai concentré mes efforts sur la configuration d'Invoice Ninja en local, en le déployant dans un conteneur Docker. Invoice Ninja propose un dépôt GitHub contenant le code source de l'application ainsi qu'une documentation détaillée pour faciliter le déploiement en local. NINJA, 2024a

Ensuite, j'ai commencé à développer mon projet API ASP.NET en suivant les principes de la Clean Architecture. J'ai configuré le Dockerfile ainsi que le fichier docker-compose pour orchestrer le déploiement de l'API et de la base de données PostgreSQL en local. Un volume a été ajouté pour garantir la persistance des données de la base de données.

Étapes de mise en œuvre :

La Clean Architecture exige une rigueur particulière dans la conception du projet, notamment pour la configuration des différentes couches. Les couches sont liées entre elles par les références explicites définies dans le fichier ".csproj" de chaque projet. Ces fichiers ".csproj" permettent de gérer les dépendances de chaque couche

Voici comment les couches sont en relation :

- **API** : dépend de la couche Application.
- **Application** : dépend de la couche Domain.
- **Domain** : ne dépend d'aucune autre couche.
- **Infrastructure** : dépend de la couche Application.

5.3.2.1 Assurer la communication entre les conteneurs

Afin de permettre la communication entre les deux conteneurs, j'ai configuré un réseau (Docker Network) dédié dans les fichiers docker-compose de chaque service. Pour vérifier que la communication entre les conteneurs était bien établie, j'ai utilisé la commande "ping" entre eux. Voici les commandes que j'ai utilisées :

```
1 docker ps
2
3
4 docker exec -it dockerfiles-app-1 ping veetamineinvoice-veetamineinvoice-1
5
6 docker exec -it veetamineinvoice-veetamineinvoice-1 ping dockerfiles-app-1
7
```

5.3.3 Étape 3 : Modélisation de la base de données

Pour la gestion de la base de données, ma première étape a été d'identifier les tables pertinentes pour mon projet. Pour cela, je me suis connecté à la base de données de Veetamine via PhpMyAdmin. J'ai commencé par repérer la table principale, "OperatorInvoices", et j'ai ensuite analysé les clés étrangères associées pour comprendre les relations entre les différentes tables. Cette exploration m'a permis de créer le modèle physique de données (MPD) suivant :

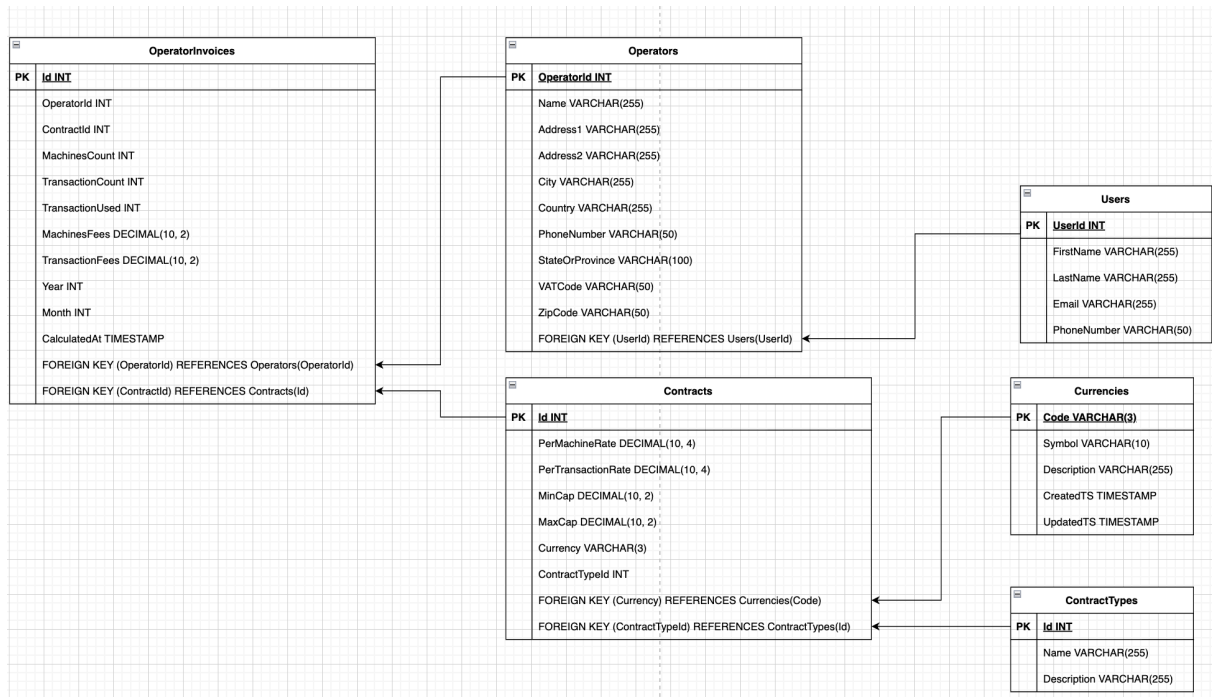


FIGURE 5.3 – Modèle physique de données (MPD) de la facturation de Veetamine
Source: de l'auteur

5.3.3.1 Création des entités

À partir de ce modèle physique de données (MPD), j'ai créé mes entités ("Operator", "OperatorInvoice", "User", etc.). Ces entités constituent la base sur laquelle repose la logique métier de l'application. Dans le cadre de la Clean Architecture, elles jouent un rôle central. Une entité peut être soit un objet comportant des méthodes, soit un ensemble de structures de données accompagnées de fonctions. Les entités représentent les concepts et les règles de base du système ODILE, 2022.

L'exemple ci-dessous présente l'entité "User", illustrant la structure de base de l'entité avec ses contraintes :

```

1
2 namespace VeetamineInvoice.Domain.Entities;
3
4 [Table("Users")]
5 public class User
    
```

```

6 {
7     [Key]
8     [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
9     public int Id { get; set; }
10
11     [Required]
12     [StringLength(127)]
13     public string Email { get; set; }
14
15 }

```

5.3.3.2 Synchronisation avec la base de données

Après avoir créé les entités, j'ai utilisé Entity Framework Core pour générer les migrations. Les migrations facilitent la synchronisation entre les modèles de données (entités) et la structure physique de la base de données, garantissant que les modifications du modèle sont correctement reflétées dans la base de données. L. MICROSOFT, 2023e

Ensuite, j'ai inséré quelques données dans ma base de données conteneurisée en utilisant un fichier SQL. Ce fichier avait été initialement généré via PhpMyAdmin, puis adapté avec l'aide de l'intelligence artificielle, ChatGPT 4o. Voici une petite documentation qui m'a été utile pour effectuer plusieurs migrations :

```

1
2 # Lister les conteneurs en cours d'exécution
3 docker ps
4
5 # Copier le fichier data.sql dans le conteneur PostgreSQL
6 docker cp /Users/maxime/Documents/hipshares/VeetamineInvoice/data.sql
   ↪ veetamineinvoicedb:/data.sql
7
8 # Accéder au conteneur PostgreSQL
9 docker exec -it veetamineinvoicedb bash
10
11 # Exécuter le script SQL dans le conteneur
12 psql -U postgres -d postgres -f /data.sql
13
14

```

5.3.4 Étape 4 : Mise en place de l'environnement de production

Le déploiement de l'application en production sur Heroku a été entièrement automatisé grâce à un pipeline CI/CD mis en place avec GitHub Actions. Les secrets et variables d'environnement nécessaires au bon fonctionnement de l'application en production sont gérés via GitHub Secrets. Un Dockerfile spécifique a été créé pour le déploiement en production, en utilisant une image de base ASP.NET Core 8.0.

Une configuration est requise dans l'interface Heroku pour activer le déploiement automatique après chaque commit sur la branche principale.

La base de données n'est pas déployée depuis l'environnement de développement. J'ai plutôt opté pour le service de base de données fourni par Heroku. Les tables de la base de données ont été créées en utilisant les migrations d'Entity Framework Core, et les données ont été insérées manuellement en me connectant à ma base de données en ligne de commande. Pour effectuer ces opérations, il est nécessaire de disposer des commandes CLI sur sa machine.

5.3.5 Étape 5 : Développement de l'API

5.3.5.1 Création de Endpoints selon la Clean Architecture

La cinquième phase du projet a été dédiée à l'implémentation des opérations CRUD pour les entités "User" et "Operator".

Les étapes suivantes ont été réalisées pour atteindre cet objectif :

1. **Création du Repository** : J'ai commencé par créer un "Repository" dans la couche Infrastructure. Ce composant est chargé de l'interaction avec la base de données, en encapsulant les opérations CRUD. Cela permet aux autres couches de l'application de manipuler les données sans se soucier des détails de persistance. Le repository est défini via une interface située dans la couche Domain.
2. **Création des Services** : Ensuite, j'ai développé des services dans la couche Application, nommés "UserService" et "OperatorService". Ces services utilisent les méthodes du repository pour exécuter des opérations sur les données. Ils encapsulent la logique métier et servent de couche intermédiaire entre les contrôleurs et les repositories. Les services sont également définis selon une interface.
3. **Création des Contrôleurs** : Enfin, j'ai mis en place les contrôleurs dans la couche API. Les contrôleurs sont responsables de la gestion des requêtes HTTP et utilisent les services pour effectuer les opérations demandées.

Pour chaque opération CRUD, l'application traverse les quatre couches de l'architecture.

5.3.5.2 Tester l'API de Invoice Ninja

Ensuite, j'avais besoin de m'assurer que je pouvais envoyer des requêtes à l'API d'Invoice Ninja. Avant de passer à l'implémentation d'un service dans mon code pour gérer ces requêtes, j'ai effectué des tests manuels avec Postman. Pour ce faire, j'ai commencé par ajouter des clients via l'interface d'Invoice Ninja, puis j'ai utilisé Postman pour récupérer ces données, en m'assurant que les identifiants dans le header étaient corrects.

J'ai également utilisé la console de Google Chrome pour examiner les schémas des requêtes lors de la création d'un client depuis l'interface d'Invoice Ninja. Bien que la documentation officielle soit disponible, certains champs n'étaient pas clairement documentés, notamment en

ce qui concerne la possibilité de laisser un champ vide ou non. Ces vérifications manuelles m'ont permis de mieux comprendre les requêtes nécessaires et de préparer le terrain pour une implémentation correcte dans mon code.

5.3.6 Étape 6 : Gestion des données clients

Lors des premières phases du projet, j'ai consulté le CTO de Cellsmaniak pour déterminer la meilleure manière de gérer les données clients entre l'API de Veetamine et Invoice Ninja. Initialement, la gestion centralisée des clients dans Invoice Ninja était proposée. Cependant, après une analyse approfondie des fonctionnalités existantes de Veetamine, telles que l'invitation de nouveaux utilisateurs et la gestion des rôles, cette approche a été jugée contre-intuitive. La dispersion de la gestion des utilisateurs sur deux applications distinctes risquait d'introduire une complexité inutile et de compromettre la maintenabilité du système.

Décision :

Pour maintenir une architecture simple et robuste, nous avons opté pour une gestion exclusive des données clients via l'API de Veetamine, avec une synchronisation future prévue entre les deux systèmes. Cette décision a été motivée par une analyse des contraintes temporelles du projet de Bachelor et la nécessité de maintenir une cohérence dans la gestion des données utilisateurs.

5.3.7 Étape 7 : Génération et envoi des factures

Actuellement, Veetamine ne facture qu'un seul client chaque mois. Pour simuler un scénario plus réaliste, j'ai généré des données de test pour plusieurs clients (deux au total).

Pour le moment, Cellsmaniak n'a pas encore défini précisément comment les clients seront facturés à l'avenir. Cependant, à ce jour, la facturation repose sur quatre services avec un prix unitaire fixe et des quantités calculées via l'API de Veetamine. Dans mon implémentation, les quantités sont directement extraites de la base de données.

ITEM	DESCRIPTION	UNIT COST	QUANTITY	LINE TOTAL
Client digital serv	Fix fee per machine per month	7.00	490.00	CHF 3'430.00
Client variable d'ig	Variable fee per machine per month	0.00	541'000.00	CHF 1'947.60
VEETAMINE fix di	Fix fee per machine per month	1.00	490.00	CHF 490.00
VEETAMINE varia	Variable fee per machine per month	0.00	541'000.00	CHF 1'298.40

FIGURE 5.4 – *Détail de la facture client*
Source: de l'auteur

Par conséquent, j'ai opté pour une approche où la structure de la facture est prédéfinie et codée en dur, tandis que les quantités de chaque service sont récupérées de manière dynamique. Si une valeur est manquante, le service sera tout de même inclus dans la facture avec une quantité de 0, ce qui entraînera un montant de 0.

```
1
2  var invoiceDto = new InvoiceNinjaDto
3      {
4          Client_id = invoiceNinjaClientId,
5          Status_id = "1",
6          Is_deleted = false,
7          Number = invoiceNumber,
8          Date = currentDate.ToString("yyyy-MM-dd"),
9          Due_date = currentDate.AddMonths(1).ToString("yyyy-MM-dd"),
10         Uses_inclusive_taxes = false,
11         Tax_name1 = taxRate.Name,
12         Tax_rate1 = taxRate.Rate,
13         Line_items =
14             [
15                 new LineItemDto
16                 {
17                     Quantity = calculatedMachinesCount,
18                     Cost = 7,
19                     Product_key = "Client digital service fee",
20                     Product_cost = 0,
21                     Notes = "Fix fee per machine per month",
22                     Discount = 0,
23                     Is_amount_discount = true,
24                     Sort_id = 0,
25                     Line_total = 7 * calculatedMachinesCount,
26                     Tax_amount = 0,
27                     Gross_line_total = 7 * calculatedMachinesCount,
28                     Type_id = 1,
29                     Tax_id = 1,
30                 },
31                 new LineItemDto
32                 {
33                     // ...
34                 }
35                 // Encore deux produits...
36             ]
37     };
38
```

5.3.7.1 Vérification de l'envoi des factures

Pour éviter de régénérer une facture déjà envoyée, il est essentiel de stocker l'information indiquant que la facture a été envoyée quelque part.

La solution que j'ai adoptée consiste à ajouter une colonne "IsInvoiceSent" de type booléen à la table "OperatorInvoices". Par défaut, cette colonne est définie sur "False". Lorsque l'API d'Invoice Ninja me confirme que la facture a été créée avec succès, je mets ce champ à "True".

Cela me permet également d'ajouter une condition supplémentaire au début du processus de génération des factures : seules les factures dont le champ "IsInvoiceSent" est défini sur "False" seront prises en compte pour être générées. Il ne sera pas nécessaire de vérifier toutes les lignes, mais uniquement celles correspondant au mois précédent.

Voici, une correction de la table "OperatorInvoices" :

```
postgres=# select * from "OperatorInvoices";
```

Id	OperatorId	ContractId	MachinesCount	TransactionCount	TransactionUsed	MachinesFees	TransactionFees	Year	Month	CalculatedAt	IsInvoiceSent
17	2	3	470	570000	570000	470	1369.1016	2024	5	2024-06-04 17:22:25.428+00	f
18	3	3	490	580000	580000	490	1369.1016	2024	5	2024-06-04 17:22:25.428+00	f
19	2	3	475	555000	555000	475	1369.1016	2024	5	2024-07-04 17:22:25.428+00	f
20	2	3	485	556000	556000	485	1369.1016	2024	6	2024-07-04 17:22:25.428+00	f
21	3	3	465	558000	558000	465	1369.1016	2024	5	2024-07-04 17:22:25.428+00	f
22	3	3	490	556000	556000	490	1369.1016	2024	6	2024-07-04 17:22:25.428+00	f
35	2	3	500	543000	543000	500	1369.1016	2024	6	2024-08-04 17:22:25.428+00	f
37	3	3	480	546000	546000	480	1369.1016	2024	6	2024-08-04 17:22:25.428+00	f
36	2	3	495	570000	570000	495	1369.1016	2024	7	2024-08-04 17:22:25.428+00	t
38	3	3	485	576000	576000	485	1369.1016	2024	7	2024-08-04 17:22:25.428+00	t

(10 rows)

FIGURE 5.5 – Table OperatorInvoices modifiée pour assurer la vérification de l'envoi des factures

Source: de l'auteur

5.3.8 Étape 8 : Automatisation de la génération des factures

Je suis sur le point de finaliser la génération des factures, mais il reste encore une étape cruciale : automatiser ce processus. En termes techniques, cela revient à mettre en place une tâche planifiée, souvent appelée "Cron Job", pour s'assurer que les factures sont générées automatiquement à des intervalles réguliers.

La notion de cron, également connue sous le nom d'expression cron, est une syntaxe utilisée pour définir des planifications basées sur l'heure et la date dans divers systèmes d'exploitations et application.

Une expression cron se compose de cinq ou six champs qui spécifient l'heure et la date auxquelles une tâche ou une commande particulière doit être exécutée. Les champs sont séparés par des espaces et représentent les valeurs suivantes GIESEL, 2023 :

*	*	*	*	*	
—	—	—	—	—	
				+-----	day of the week (0 - 6) (Sunday=0)
			+-----		month (1 - 12)
		+-----			day of the month (1 - 31)
	+-----				hour (0 - 23)
+-----					minute (0 - 59)

FIGURE 5.6 – Explication de l'expression cron

Source: <https://steven-giesel.com/blogPost/fb1ce2ab-dd27-43ed-aaab-077adf2d15cd>

J'ai constaté que les bibliothèques les plus régulièrement citées étaient pour la gestion des cron job Hangfire et Quartz.NET. Initialement, j'avais choisi Hangfire par sa nature plus intuitive et sa facilité d'utilisation. De plus, Hangfire disposait d'un dashboard qui permet de visualiser les tâches avec leur status. J'avais trouvé cette intégration très pratique, car depuis le dashboard je pouvais trigger les tâches manuellement.

Cependant, après avoir déployé mon application sur Heroku, j'ai rencontré une erreur que je n'ai pas pu résoudre entièrement. Il semble que le problème était lié à l'incapacité d'Heroku à gérer correctement les routes de Swagger et du tableau de bord Hangfire. Malheureusement, je n'ai pas pris de capture d'écran de l'erreur pour en partager les détails. Face à cette situation, j'ai décidé de passer à Quartz.NET. Ce choix s'explique par le fait que la configuration de Hangfire n'avait pas encore demandé beaucoup de temps, et il m'a semblé plus judicieux de changer de bibliothèque plutôt que de passer du temps à déboguer l'erreur.

5.3.9 Étape 9 : Test et déploiement

L'implémentation des tests ont été réalisés en phase terminal de ce projet. Je me suis concentré des tester les services qui ont une grande importance pour le fonctionnement de l'application comme "InvoiceGenerationService". Je n'ai implémenté que des tests unitaires.

Concernant le déploiement, assez vite mon fichier YAML était opérationnel, car c'était une de mes priorités importantes pour le début du projet. Je l'ai ajusté en fin de projet afin qu'il intègre la vérification des tests unitaires.

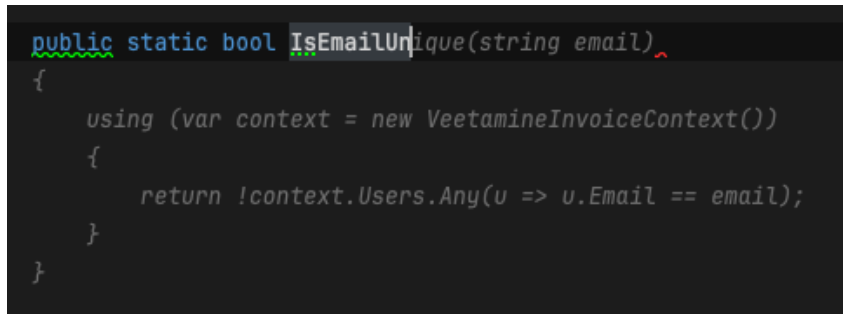
5.4 Utilisation de l'intelligence artificielle

Pour ce projet, j'ai utilisé deux intelligences artificielles : ChatGPT 4o et Copilot de GitHub.

5.4.1 Copilot

En tant qu'étudiant à la HES-SO de Sierre, nous avons la possibilité de demander une licence professionnelle GitHub Education. Cette licence m'a permis d'accéder à GitHub Copilot, un outil qui s'est révélé particulièrement utile pour l'écriture de code.

Au début du projet, alors que j'implémentais des méthodes relativement simples, l'auto-complétion de Copilot m'a fait gagner un temps précieux. Parfois, il me suggérait des solutions auxquelles je n'avais pas pensé, mais qui s'avéraient parfaitement adaptées au contexte. Cela m'a incité à être curieux et à explorer pourquoi ces solutions fonctionnaient. En somme, j'ai utilisé Copilot non seulement comme un assistant de développement, mais aussi comme un outil d'apprentissage et de formation.

A screenshot of a code editor showing a C# method named `IsEmailUnique`. The code is as follows:

```
public static bool IsEmailUnique(string email)
{
    using (var context = new VeetamineInvoiceContext())
    {
        return !context.Users.Any(u => u.Email == email);
    }
}
```

The text `IsEmailUn` is highlighted, and the word `ique` is being auto-completed by GitHub Copilot, indicated by a small red squiggly line under the `ique` part of the method name.

FIGURE 5.7 – Auto-complétion de GitHub Copilot
Source: de l'auteur

5.4.2 ChatGPT 4o

Grâce à l'accès premium à ChatGPT-4.0 via une licence entreprise, j'ai pu utiliser cet outil pour m'assister dans plusieurs aspects de mon travail.

ChatGPT-4.0 m'a été particulièrement utile pour répondre à des questions précises et m'aider à formuler des phrases. Je trouve que ChatGPT-4.0 excelle lorsqu'il s'agit de traiter une grande quantité d'informations. Par exemple, face à certaines erreurs Docker avec des centaines de lignes de logs, j'ai simplement copié ces logs dans ChatGPT-4.0 et il m'a fourni une réponse précise.

Je l'ai également utilisé pour traduire du code d'un langage de programmation ou de requête à un autre. Par exemple, PhpMyAdmin m'avait généré un script SQL pour insérer des données dans ma base de données, mais ce script n'était pas compatible avec PostgreSQL. J'ai donc utilisé ChatGPT-4.0 pour convertir ce script en une version exécutable sous PostgreSQL, ce qui m'a permis de surmonter cet obstacle.

6 | Réalisation

6.1 Le service InvoiceGenerationService

Le service "InvoiceGenerationService" est responsable de la génération des factures pour les opérateurs. La classe "InvoiceGenerationService" suit les principes de la Clean Architecture, avec une forte utilisation de l'injection de dépendances.

Initialement, j'avais construit la méthode "GenerateInvoicesAsync" de manière monolithique. Après avoir confirmé son bon fonctionnement à travers des tests d'utilisation, je l'ai segmentée en plusieurs méthodes plus petites et spécialisées, afin de respecter les principes de séparation des responsabilités. Cette approche m'a permis de tester plus facilement chaque partie de la méthode et d'améliorer la maintenabilité du code.

Pour une meilleure lisibilité, les extraits de code sont disponibles en annexe. Toutefois, j'ai inclus les tests unitaires spécifiques pour les méthodes concernées.

6.1.1 Méthode GenerateInvoicesAsync

La méthode "GenerateInvoicesAsync" (voir l'annexe I.1) est le cœur du service. Elle gère l'ensemble du processus de création de factures. Voici comment elle fonctionne :

- **Récupération des dates** : Elle commence par calculer les dates nécessaires, comme le mois actuel, le mois précédent, et deux mois en arrière.
- **Récupération des opérateurs concernés** : Ensuite, elle identifie les opérateurs pour lesquels des factures doivent être générées.
- **Traitement des factures par opérateur** : Pour chaque opérateur, elle récupère les factures des mois précédents et calcule les valeurs nécessaires pour la nouvelle facture.
- **Préparation et envoi des factures** : Enfin, elle prépare la facture et l'envoie à Invoice Ninja, le logiciel utilisé pour gérer les factures.

6.1.2 Méthode CalculateDates

Cette méthode sert à calculer les dates importantes pour la génération des factures, comme le mois en cours, le mois précédent, et deux mois en arrière. Ces dates sont ensuite utilisées pour comparer et calculer les valeurs des factures. (voir l'annexe I.2)

Test unitaire : Ce test a pour objectif de vérifier que la méthode retourne correctement les dates pertinentes (mois actuel, mois précédent et deux mois avant) en fonction d'une date donnée, en l'occurrence, le mois de juin.

```

1  public void Calculate-
   ↪   Dates_WithCurrentMonthJune_Should_Return_CorrectPreviousAndTwoMonthsBefore()
2  {
3      // Arrange
4      var testDate = new DateTime(2024, 6, 15); // Mois de juin (6)
5
6      var service = new InvoiceGenerationService(null, null, null, null, null, null);
7
8      // Act
9      var (currentDate, currentYear, currentMonth, previousMonth, previousYear,
   ↪       twoMonthsBefore, twoMonthsBeforeYear) = service.CalculateDates(testDate);
10
11     // Assert
12     Assert.Equal(6, currentMonth);
13     Assert.Equal(5, previousMonth);
14     Assert.Equal(4, twoMonthsBefore);
15     Assert.Equal(2024, currentYear);
16     Assert.Equal(2024, previousYear);
17     Assert.Equal(2024, twoMonthsBeforeYear);
18 }

```

6.1.3 Méthode ProcessOperatorInvoices

Cette méthode s'occupe du traitement des factures pour un opérateur donné. Elle suit plusieurs étapes : (voir l'annexe I.3)

- Elle commence par récupérer les factures des deux derniers mois pour l'opérateur en question.
- Ensuite, elle vérifie que le nombre de factures récupérées pour les deux derniers mois est conforme aux attentes (une facture pour le mois précédent et deux factures pour deux mois avant).
- Si c'est le cas, elle calcule ces écarts et les utilise pour déterminer les valeurs de la nouvelle facture.
- Enfin, elle prépare les données nécessaires pour envoyer la facture à Invoice Ninja en utilisant la méthode "PrepareInvoiceDto".

6.1.4 Méthode CalculateInvoiceValues

Cette méthode prend les factures des mois précédents et calcule les valeurs nécessaires pour la facture actuelle, comme les différences dans les transactions utilisées ou le nombre de machines facturées. (voir l'annexe I.4)

Test unitaire : Ce test vérifie que la méthode calcule correctement les valeurs nécessaires à la génération d'une nouvelle facture, en se basant sur les données des factures des mois précédents.

```
1 public void CalculateInvoiceValues_Should_Return_Correct_Values()
2 {
3     // Arrange
4     var previousMonthInvoice = new OperatorInvoice { TransactionUsed = 100,
5     ↪ MachinesCount = 5 };
6     var firstTwoMonthsBeforeInvoice = new OperatorInvoice { TransactionUsed = 90,
7     ↪ MachinesCount = 5 };
8     var secondTwoMonthsBeforeInvoice = new OperatorInvoice { TransactionUsed = 110,
9     ↪ MachinesCount = 6 };
10
11     // Act
12     var (calculatedTransactionUsed, calculatedMachinesCount) =
13     ↪ _service.CalculateInvoiceValues(previousMonthInvoice,
14     ↪ firstTwoMonthsBeforeInvoice, secondTwoMonthsBeforeInvoice);
15
16     // Assert
17     Assert.Equal(120, calculatedTransactionUsed);
18     Assert.Equal(6, calculatedMachinesCount);
19 }
```

6.1.5 Méthode PrepareInvoiceDto

Cette méthode construit un objet DTO (Data Transfer Object), qui contient toutes les informations nécessaires pour envoyer la facture à Invoice Ninja. Elle vérifie que l'opérateur a un ID client valide pour Invoice Ninja, récupère le taux de TVA applicable, et crée les articles de la facture à partir des valeurs précédemment calculées. (voir l'annexe I.5)

Test unitaire : Le test doit valider que PrepareInvoiceDto prépare correctement les données nécessaires à l'envoi de la facture, en s'assurant que les informations critiques comme l'ID du client, le numéro de facture, et le taux de taxe sont bien présents et corrects.

```
1 public async Task PrepareInvoiceDto_Should_Return_Valid_InvoiceNinjaDto()
2 {
3     // Arrange
4     var calculatedValues = (calculatedTransactionUsed: 120m, calculatedMachinesCount:
5     ↪ 6);
6     var operatorEntity = new Operator { InvoiceNinjaClientId = "Client123" };
7     _mockOperatorService.Setup(s =>
8     ↪ s.GetOperatorByIdAsync(It.IsAny<int>())).ReturnsAsync(operatorEntity);
9     _mockSettingInvoiceNinjaService.Setup(s =>
10    ↪ s.GetNextInvoiceNumberAsync()).ReturnsAsync("INV123");
11    _mockTaxService.Setup(s => s.GetTaxRateAsync("TVA Suisses")).ReturnsAsync(new
12    ↪ TaxRateDto { Name = "TVA Suisses", Rate = 8.1m });
13
14    // Act
15    var result = await _service.PrepareInvoiceDto(1, calculatedValues, DateTime.Now);
16
17    // Assert
18    Assert.NotNull(result);
19    Assert.Equal("Client123", result.Client_id);
20    Assert.Equal("INV123", result.Number);
21    Assert.Equal(8.1m, result.Tax_rate1);
22 }
```

6.1.6 Méthode GenerateLineItems

Cette méthode génère les articles de la facture, appelés "line items". Ces articles incluent des détails comme le nombre de machines facturées ou les transactions effectuées, et sont calculés à partir des données disponibles. (voir l'annexe I.2)

6.1.7 Méthode SendInvoice

Cette méthode envoie la facture préparée à Invoice Ninja. Si l'envoi est réussi, elle marque la facture du mois précédent comme envoyée en mettant à jour le champ "IsInvoiceSent" à "true". (voir l'annexe I.7)

Test unitaire : Pour cette méthode, j'ai testé que si l'envoi de la facture est un succès, le champ "IsInvoiceSent" est à "true".

```

1  public async Task SendInvoice_Should_Update_InvoiceStatus_If_Success()
2  {
3      // Arrange
4      var previousMonthInvoice = new OperatorInvoice { Id = 1, IsInvoiceSent = false };
5      var invoiceDto = new InvoiceNinjaDto();
6      var invoiceResponse = new InvoiceResponseDto { IsSuccess = true };
7      _mockInvoiceNinjaClientService.Setup(s =>
8          s.CreateInvoiceAsync(invoiceDto)).ReturnsAsync(invoiceResponse);
9
10     // Act
11     await _service.SendInvoice(previousMonthInvoice, invoiceDto);
12
13     // Assert
14     Assert.True(previousMonthInvoice.IsInvoiceSent);
15     _mockOperatorInvoiceService.Verify(s => s.UpdateInvoiceAsync(previousMonthInvoice),
16         Times.Once);
17 }
```

6.1.8 Conclusion

Bien que plusieurs cas aient été testés, d'autres tests pourraient être ajoutés, comme s'assurer qu'une facture déjà envoyée n'est pas à nouveau envoyée. Faute de temps, tous les tests n'ont pas pu être couverts.

6.2 Cron Job

Pour configurer un cron job avec Quartz.NET, il est d'abord nécessaire de configurer le(s) Trigger(s). Dans mon cas, j'ai choisi de configurer deux triggers. Le premier est exécuté tous les premiers jours du mois à 2h du matin. Le second est configuré en tant que sécurité. En effet, si, pour une raison ou une autre, il n'y a aucune nouvelle donnée à traiter à 2h du matin, un deuxième trigger sera automatiquement déclenché à 16h. Cette configuration est à implémenter dans le fichier "Program.cs". QUARTZ.NET, 2024

Chapitre 6. Réalisation

Dans le scénario où le premier trigger est exécuté et génère les factures avec succès, le deuxième trigger sera appelé mais pas exécuté, car j'ai mis en place un suivi de l'état du "Job". Ce suivi est basé sur la combinaison du mois et de l'année en cours.

```
1      q.AddJob<InvoiceGenerationJob>(opts => opts.WithIdentity("GenerateInvoices", "group1"));
2
3
4      // Schedule the first job
5      q.AddTrigger(opts => opts
6          .ForJob("GenerateInvoices", "group1")
7          .WithIdentity("GenerateInvoicesFirst", "group1")
8          .StartNow()
9          .WithCronSchedule("0 0 02 1 * ?")); // Run at 02:00 AM on the first day of every
        ↪ month
10
11     // Schedule the second job
12     q.AddTrigger(opts => opts
13         .ForJob("GenerateInvoices", "group1")
14         .WithIdentity("GenerateInvoicesTriggerSecond", "group1")
15         .StartNow()
16         .WithCronSchedule("0 0 16 1 * ?")); // Run at 4:00 PM on the first day of every
        ↪ month
17
```

Ensuite, il est nécessaire de configurer un service en implémentant l'interface "IJob" de Quartz.NET. Cette interface définit une méthode, "Execute", qui est automatiquement appelée par Quartz.NET chaque fois que le job est déclenché. Cette méthode encapsule la logique principale du job. Dans mon cas, le job invoque de manière asynchrone le service "InvoiceGenerationService" pour générer les factures.

```
1      namespace VeetamineInvoice.Applications.Services;
2
3
4      public class InvoiceGenerationJob : IJob
5      {
6          private readonly IInvoiceGenerationService _invoiceGenerationService;
7          private readonly ILogger<InvoiceGenerationJob> _logger;
8
9          public InvoiceGenerationJob(IInvoiceGenerationService invoiceGenerationService,
        ↪      ILogger<InvoiceGenerationJob> logger)
10         {
11             _invoiceGenerationService = invoiceGenerationService;
12             _logger = logger;
13         }
14
15         public async Task Execute(IJobExecutionContext context)
16         {
17             string currentMonthYear = DateTime.Now.ToString("MM-yyyy");
18
19             if (JobStatusService.IsInvoiceGenerationSuccessful(currentMonthYear))
20             {
21                 _logger.LogInformation("InvoiceGenerationJob skipped as it has already been
        ↪      successfully executed for this month.");
22                 return;
23             }
24         }
25     }
26
```

```

24         _logger.LogInformation("InvoiceGenerationJob started.");
25     try
26     {
27         await _invoiceGenerationService.GenerateInvoicesAsync();
28         _logger.LogInformation("InvoiceGenerationJob executed successfully.");
29
30         // Update the success indicator for this month
31         JobStatusService.SetInvoiceGenerationSuccess(currentMonthYear, true);
32     }
33     catch (Exception ex)
34     {
35         _logger.LogError(ex, "Error executing InvoiceGenerationJob.");
36     }
37 }
38 }
39 }
40

```

6.2.1 Persistance des données Quartz.NET

Pour assurer la persistance des données dans Quartz.NET, notamment les jobs, les triggers et l'état des jobs, j'ai dû ajouter de nouvelles tables à ma base de données. Quartz.NET dispose d'un dépôt GitHub qui fournit des scripts SQL spécifiques pour la création de ces tables, adaptés à différents types de bases de données. QUARTZ.NET, 2023

```

1
2 // Fichier Program.cs
3
4 q.UsePersistentStore(options =>
5 {
6     options.UsePostgres(connectionString);
7     options.UseNewtonsoftJsonSerializer();
8     options.UseClustering();
9 });
10

```

6.2.2 Contrôler le bon fonctionnement du Cron Job

Afin de pouvoir tester le fonctionnement de mon Cron Job, j'ai créé un contrôleur nommé "CronJobController". Ce contrôleur contient une action "TriggerJob" qui permet de déclencher manuellement un job en fonction de son nom. Cette action vérifie si le trigger existe, récupère le job associé et le déclenche. Si le trigger n'existe pas, un message d'erreur est renvoyé.

```

1
2 [ApiController]
3 [Route("api/[controller]")]
4 [ApiVersion("1.0")]
5
6 public class CronJobController : ControllerBase
7 {
8     private readonly ISchedulerFactory _schedulerFactory;
9
10    public CronJobController(ISchedulerFactory schedulerFactory)

```

```
11     {
12         _schedulerFactory = schedulerFactory;
13     }
14
15     /// <summary>
16     /// Test CronJob Demo: First jobName -> "GenerateInvoicesFirst", Second jobName ->
17     /// ↪ "GenerateInvoicesSecond"
18     /// </summary>
19     [HttpPost("Trigger/{triggerName}")]
20     public async Task<IActionResult> TriggerJob(string triggerName)
21     {
22         var scheduler = await _schedulerFactory.GetScheduler();
23         var triggerKey = new TriggerKey(triggerName, "group1");
24
25         if (await scheduler.CheckExists(triggerKey))
26         {
27             var trigger = await scheduler.GetTrigger(triggerKey);
28             var jobKey = trigger.JobKey;
29             await scheduler.TriggerJob(jobKey);
30             return Ok($"Trigger {triggerName} executed job {jobKey.Name}
31             ↪ successfully.");
32         }
33         return NotFound($"Trigger {triggerName} not found.");
34     }
35 }
36
37
```

6.3 Middleware - Rate Limiting

6.3.1 OWASP

L'Open Web Application Security Project (OWASP) est une organisation internationale à but non lucratif dédiée à la sécurité des applications web. Un des principes fondamentaux de l'OWASP est que tout son contenu soit disponible gratuitement et facilement accessible sur son site web, permettant à chacun d'améliorer la sécurité de ses propres applications.

Le Top 10 de l'OWASP est un rapport régulièrement mis à jour qui met en lumière les préoccupations en matière de sécurité des applications web, en se concentrant sur les 10 risques les plus critiques. Il existe également un Top 10 spécifique aux API. CLOUDFLARE, 2024

Dans le document de 2019, le risque intitulé "Lack of Resources and Rate Limiting" a retenu mon attention car l'absence de mise en pratique de ce principe peut entraîner un déni de service (DoS), rendant l'API insensible voire indisponible. OWASP, 2019

6.3.2 Rate Limiting middleware ASP.NET Core

Microsoft propose un middleware ASP.NET Core pour gérer les limiteurs de débit, offrant plusieurs options parmi lesquelles j'ai choisi le "Sliding Window Rate Limiter".

Le "Sliding Window Rate Limiter" est un algorithme de limitation de débit qui divise le temps en fenêtres de durée fixe, chacune étant subdivisée en segments. Chaque fenêtre peut contenir un nombre maximum de requêtes autorisées. Lorsqu'une requête est reçue, le middleware vérifie si le nombre de requêtes dans la fenêtre actuelle est inférieure au maximum autorisé. Si c'est le cas, la requête est acceptée. Sinon, une réponse 429 Too Many Requests est renvoyée.

Ce mécanisme permet de mieux répartir les requêtes dans le temps et d'éviter les pics soudains de trafic susceptibles de surcharger l'API. MICROSOFT, 2023

6.3.3 Configuration

Le code JSON représenté ci-dessous est la configuration de mon limiter de débit. Cette configuration se trouve dans le fichier "appsettings.json" de mon projet.

```
"MyRateLimitOptions":{  
  "PermitLimit": 100,  
  "Window": 60,  
  "SegmentsPerWindow": 10,  
  "QueueLimit": 5  
},
```

- **PermitLimit** : 100 - Le nombre maximum de requêtes autorisées dans une fenêtre de temps.
- **Window** : 60 secondes - La durée totale de chaque fenêtre de temps.
- **SegmentsPerWindow** : 10 - La fenêtre de temps est divisée en 10 segments.
- **QueueLimit** : 5 - Le nombre maximum de requêtes qui peuvent être mises en file d'attente une fois la limite de requêtes atteinte.

En somme cette configuration me permet de faire jusqu'à 100 requêtes dans une fenêtre de 60 secondes. Une fois cette limite atteinte, jusqu'à 5 requêtes supplémentaires peuvent être mises en file d'attente. Les requêtes supplémentaires après ces 105 requêtes entraîneront une réponse 429.

Voici une représentation à l'aide d'un tableau :

Report : 15 (45 - 40 + 10).

6.3.4 Implémentation du middleware

Depuis la documentation officielle de Microsoft, l'algorithme proposé est prêt à l'emploi. Ce middleware est à implémenter dans le fichier "Program.cs".

```
1  
2 var myOptions = new MyRateLimitOptions(); // Créer une instance de MyRateLimitOptions
```

Temps (s)	Disponible	Pris	Recyclé à partir des expirées	Report
0	100	20	0	80
10	80	30	0	50
20	50	40	0	10
30	10	30	20	0
40	0	10	30	20
50	20	10	40	50
60	50	35	30	45
70	45	40	10	15

TABLE 6.1 – *Tableau des requêtes disponibles, prises, recyclées et reportées au fil du temps*

```

3  builder.Configuration.GetSection(MyRateLimitOptions.MyRateLimit).Bind(myOptions);
4  var slidingPolicy = "sliding";
5
6  // Ajout du service de limitation de débit
7  builder.Services.AddRateLimiter(options =>
8  {
9      options.AddSlidingWindowLimiter(policyName: slidingPolicy, config =>
10     {
11         config.PermitLimit = myOptions.PermitLimit;
12         config.Window = TimeSpan.FromSeconds(myOptions.Window);
13         config.SegmentsPerWindow = myOptions.SegmentsPerWindow;
14         config.QueueProcessingOrder = QueueProcessingOrder.OldestFirst;
15         config.QueueLimit = myOptions.QueueLimit;
16     });
17 });
18

```

6.3.5 Contrôler le bon fonctionnement du middleware

Pour vérifier le bon fonctionnement du middleware de limitation de débit, j'ai implémenté un contrôleur nommé "RateLimitedController". Ce contrôleur contient une action "Get" qui simule une requête GET. J'ai délibérément réduit la limite de requêtes à 5 afin de pouvoir tester et observer le comportement du middleware de manière efficace.

```

    "MyRateLimitOptions": {
        "PermitLimit": 5,
        "Window": 10,
        "SegmentsPerWindow": 2,
        "QueueLimit": 2
    }

```

6.4 Déploiement de l'API Conteneurisé dans Heroku

La capture d'écran ci-dessous démontre qu'après 5 requêtes, l'API se met en attente pour traiter les requêtes supplémentaires. Étant donné que la fenêtre de temps est de 10 secondes, divisée en 2 segments de 5 secondes chacun, je dois attendre 5 secondes pour que le segment actuel expire et que la limite soit réinitialisée. Après, je peux à nouveau effectuer jusqu'à 5 appels avant que la limitation ne soit de nouveau appliquée.

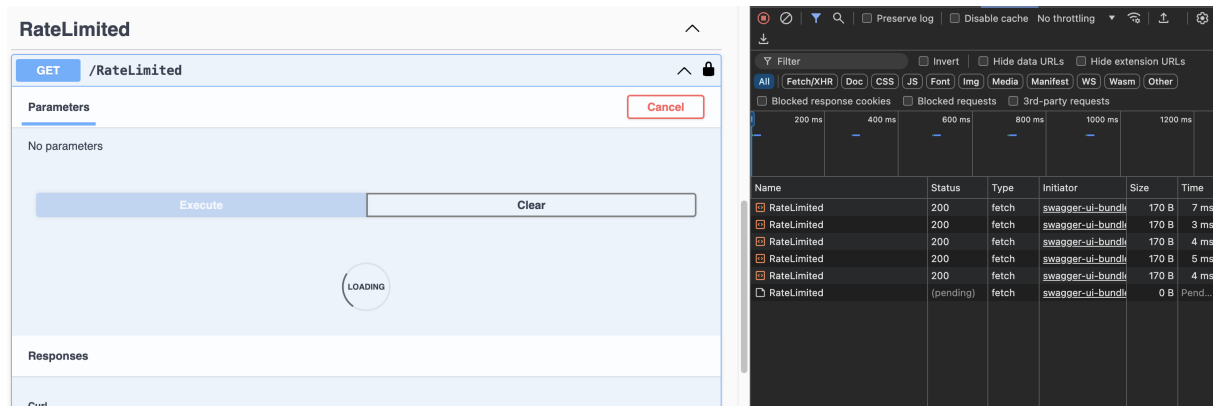


FIGURE 6.1 – Test du endpoint *RateLimited*
Source: de l'auteur

6.4 Déploiement de l'API Conteneurisé dans Heroku

6.4.1 Image Docker

Le Dockerfile définit un ensemble de directives pour créer une image contenant notre application ASP.NET Core. Cette image Docker peut ensuite être utilisée pour déployer l'application dans l'environnement de production.

Le processus commence par l'utilisation de l'image officielle du runtime ASP.NET Core 8.0, qui sert de base pour l'application. Je configure ensuite l'environnement de travail et j'expose le port 80, qui est le port par défaut pour les applications web.

```
1 FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
2 WORKDIR /app
3 EXPOSE 80
```

Ensuite, j'utilise l'image officielle du SDK .NET 8.0 pour construire l'application. Je copie les fichiers de solution et de projet nécessaires, ainsi que les différentes couches de l'application (API, Applications, Domain, Infrastructure), dans le conteneur. Puis, je restaure les dépendances NuGet pour préparer la construction de l'application.

```
1 FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
2 WORKDIR /src
3
4 COPY ["VeetamineInvoice.sln", "."]
5 COPY ["VeetamineInvoice.API/VeetamineInvoice.API.csproj", "VeetamineInvoice.API/"]
```

Chapitre 6. Réalisation

```
6 COPY ["VeetamineInvoice.Applications/VeetamineInvoice.Applications.csproj",  
  ↪ "VeetamineInvoice.Applications/"]  
7 COPY ["VeetamineInvoice.Domain/VeetamineInvoice.Domain.csproj",  
  ↪ "VeetamineInvoice.Domain/"]  
8 COPY ["VeetamineInvoice.Infrastructure/VeetamineInvoice.Infrastructure.csproj",  
  ↪ "VeetamineInvoice.Infrastructure/"]  
9  
10 RUN dotnet restore "VeetamineInvoice.API/VeetamineInvoice.API.csproj"
```

Puis, je compile l'application en mode Release et je la copie dans un dossier spécial à l'intérieur du conteneur. Enfin, j'utilise l'image de base pour la dernière étape, où je transfère l'application dans le conteneur final et configure le point de démarrage pour qu'elle se lance automatiquement lorsque le conteneur démarre.

```
1 COPY . .  
2 WORKDIR "/src/VeetamineInvoice.API"  
3 RUN dotnet build "VeetamineInvoice.API.csproj" -c Release -o /app/build  
4  
5 FROM build AS publish  
6 RUN dotnet publish "VeetamineInvoice.API.csproj" -c Release -o /app/publish  
  ↪ /p:UseAppHost=false  
7  
8 FROM base AS final  
9 WORKDIR /app  
10 COPY --from=publish /app/publish .  
11  
12 ENTRYPOINT ["dotnet", "VeetamineInvoice.API.dll"]
```

6.4.2 Pipeline CI/CD avec GitHub Actions

Pour que GitHub Actions déclenche un pipeline, une structure spécifique doit être respectée.

```
1 /.github  
2   /workflows  
3     deploy.yml  
4
```

Le pipeline CI/CD est configuré pour automatiser le processus d'intégration continue et de déploiement continu de l'application VeetamineInvoice. À chaque modification du code source sur la branche principale, le pipeline s'exécute automatiquement.

La première phase consiste à intégrer les changements de code. Le code source est d'abord récupéré depuis le dépôt GitHub, puis l'environnement .NET Core 8.0 est configuré pour préparer la construction de l'application. Les dépendances nécessaires sont restaurées via NuGet, et le projet est ensuite construit. Une fois la construction terminée, des tests unitaires sont exécutés pour vérifier que l'application fonctionne correctement.

6.4 Déploiement de l'API Conteneurisé dans Heroku

Après avoir validé l'intégration, la seconde phase se concentre sur le déploiement. Le code est de nouveau récupéré et l'environnement .NET Core est reconfiguré. Ensuite, une image Docker de l'application est construite à partir du Dockerfile fourni. Cette image est ensuite poussée vers le registre de conteneurs de Heroku après authentification via une clé API. Enfin, l'application est déployée sur Heroku en utilisant l'image Docker précédemment créée, assurant ainsi que la version la plus récente de l'application est mise en production.

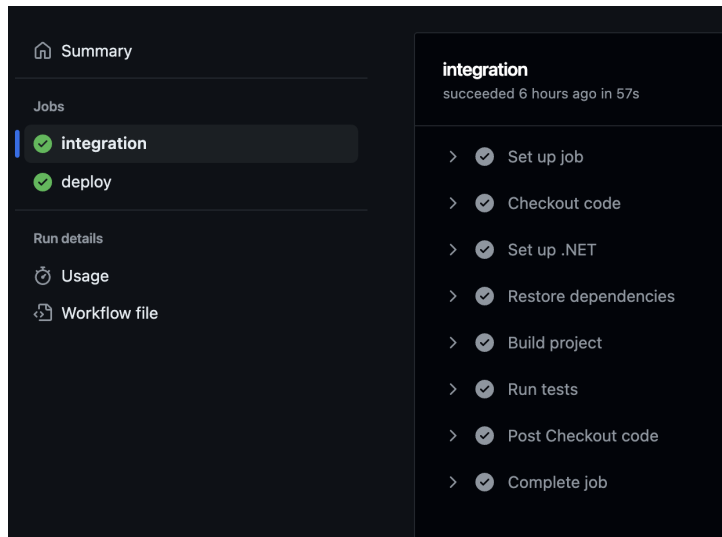


FIGURE 6.2 – *Étape d'intégration dans le pipeline*
Source: de l'auteur

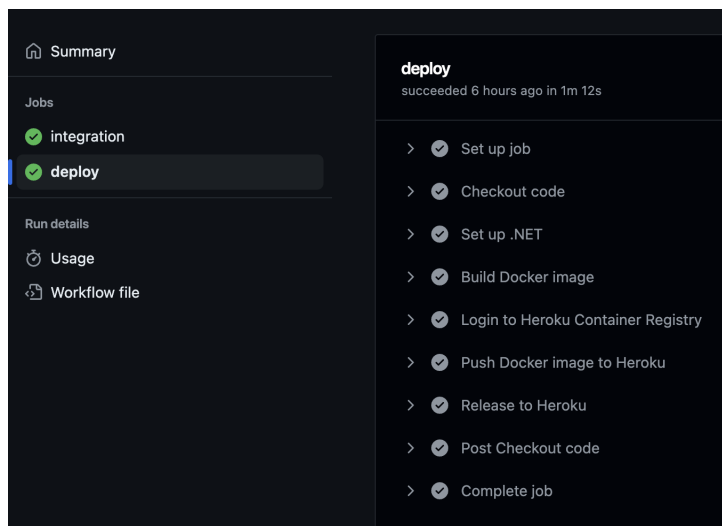


FIGURE 6.3 – *Étape de déploiement dans le pipeline*
Source: de l'auteur

Cette réalisation a été rendue possible grâce à la lecture de cet article. QUARTZ.NET, 2023

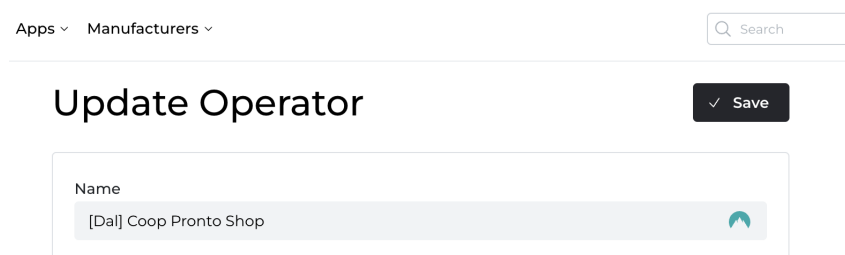
7 | Résultats

7.1 Mapper les données

La personne en charge de la gestion des contrats clients chez Veetamine doit se poser la question suivante : "Qui est la personne de référence de cet opérateur pour que je puisse envoyer les factures ?". Il est donc nécessaire d'établir une relation entre la table "Users" et "Operateurs".

Cette opération sera bien entendu effectuée via une interface graphique, notamment depuis le tableau de bord de Veetamine.

Prenons par exemple l'image ci-dessous, qui est une capture d'écran de l'édition d'un opérateur via le tableau de bord de Veetamine. Actuellement, seul le nom de l'opérateur peut être modifié via l'interface. Cependant, une liste d'utilisateurs liés à cet opérateur est disponible. À partir de cette liste, il est essentiel de déterminer qui est la personne de référence en charge de la gestion des factures pour l'entreprise.



The screenshot shows a web interface for updating an operator. At the top, there are navigation links 'Apps' and 'Manufacturers', and a search bar. The main heading is 'Update Operator'. To the right of the heading is a 'Save' button with a checkmark icon. Below the heading is a form with a label 'Name' and a text input field containing '[Dal] Coop Pronto Shop'. There is a small blue circular icon with a white arrow in the bottom right corner of the input field.

FIGURE 7.1 – Mise à jour de l'opérateur depuis le dashboard de Veetamine
Source: de l'auteur

C'est là que mon travail intervient. J'ai développé le point de terminaison `/api/Operator/id/ReferencePerson` qui permet de créer un client dans Invoice Ninja. À l'avenir, le code frontend de Veetamine pourra utiliser ce point de terminaison pour synchroniser les données.

POST /api/Operators/{id}/ReferencePerson This POST also creates a client in Invoice Ninja

Parameters

Name	Description
id * required integer(\$int32) (path)	34

Request body

```
{
  "operatorId": 34,
  "userId": 34
}
```

FIGURE 7.2 – Point de terminaison pour attribuer une personne de référence
Source: de l'auteur

L'image ci-dessous démontre clairement l'intégration entre les deux applications, puisque l'ID du client Invoice Ninja est désormais stockée dans l'entité "Operator".

```
{
  "operatorId": 34,
  "name": "Entreprise",
  "address1": "Rte de la gare 44",
  "address2": "string",
  "city": "Bulle",
  "country": "Suisse",
  "phoneNumber": "0798887766",
  "stateOrProvince": "Fribourg",
  "vatCode": "VAT9804233",
  "zipCode": "1630",
  "referencePersonId": 34,
  "invoiceNinjaClientId": "qM7e5WZb2v",
  "operatorInvoices": []
}
```

FIGURE 7.3 – Schéma de données de "Opérateur" prouvant le mappage
Source: de l'auteur

7.1.1 Résultat du mapping

Voici le résultat obtenu dans Invoice Ninja après l'exécution de la méthode POST "/api/Operator/id/ReferencePerson".

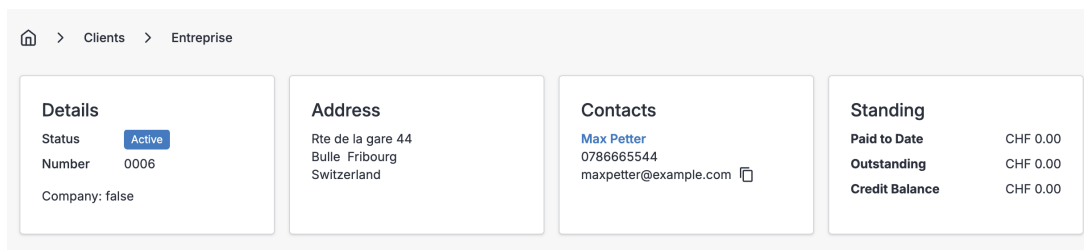


FIGURE 7.4 – Détail d'un client depuis l'interface de Invoice Ninja
Source: de l'auteur

Les bénéfices de cette implémentation sont les suivants :

- **Gain de temps :** Les informations relatives aux clients et aux opérateurs ne sont saisies qu'une seule fois par un utilisateur, puis ces données sont automatiquement transmises entre les deux applications.
- **Centralisation :** Cette solution renforce l'idée que toutes les données sont gérées à partir d'une seule plateforme, offrant ainsi une meilleure cohérence et une gestion plus efficace.

7.2 Génération des factures

Le point de terminaison `"/api/CronJob"` a été créé à des fins de test pour le développement. Cependant, il est tout à fait envisageable que le frontend de Veetamine puisse déclencher la génération à des fins d'urgence par exemple.

Pour que celui-ci fonctionne correctement, il est nécessaire de passer son nom de Job en paramètre. De plus, il faut s'assurer que les dernières données n'ont pas déjà été traitées sinon le Job ne sera pas exécuté.

CronJob

POST `/api/CronJob/Trigger/{jobName}` Test CronJob Demo: First jobName -> "GeneratelInvoicesFirst",

Parameters

Name	Description
jobName * required string (path)	GeneratelInvoicesFirst

Execute

FIGURE 7.5 – Point de terminaison pour déclencher le Cron Job manuellement
Source: de l'auteur

L'illustration ci-dessous montre que deux factures ont été générées pour des opérateurs différents, prouvant que le code peut gérer plusieurs opérateurs simultanément.

	STATUS	NUMBER	CLIENT	AMOUNT	BALANCE	DATE	DUE DATE	
<input type="checkbox"/>	Draft	1007-2024	Eversys	CHF 8'023.18	CHF 0.00	14/Aug/2024	14/Sep/2024	More Actions
<input type="checkbox"/>	Draft	1008-2024	Franke	CHF 7'778.88	CHF 0.00	14/Aug/2024	14/Sep/2024	More Actions

FIGURE 7.6 – Listes de factures depuis Invoice Ninja
Source: de l'auteur

Enfin, voici un exemple de facture générée pour un opérateur spécifique.

TestTB SA		Switzerland		
INVOICE				
Invoice Number	1008-2024	Franke		
Invoice Date	14/Aug/2024	0005		
Due Date	14/Sep/2024	Industrie		
Invoice Total	CHF7'778.88	Berne, Berne		
Balance Due	CHF7'778.88	Switzerland		
		nico@veetamine.com		
Item	Description	Unit Cost	Quantity	Line Total
Client digital service fee	Fix fee per machine per month	CHF7.00	475	CHF3'325.00
Client variable digital service fee	Variable fee per machine per month	CHF0.0036	566'000	CHF2'037.60
VEETAMINE fix digital service fee	Fix fee per machine per month	CHF1.00	475	CHF475.00
VEETAMINE variable digital service fee	Variable fee per machine per month	CHF0.0024	566'000	CHF1'358.40
		Net		CHF7'196.00
		Subtotal		CHF7'196.00
		TVA Suisses 8.1%		CHF582.88
		Total		CHF7'778.88
		Paid to Date		CHF0.00
		Balance Due		CHF7'778.88

FIGURE 7.7 – *Détail d'une facture en PDF pour un client*
Source: de l'auteur

Les bénéfices de cette implémentation sont les suivants :

- **Gain de temps** : Bien que je n'aie pas encore mesuré précisément le gain de temps par rapport à l'ancienne méthode, il est clair que la génération des factures est instantanée une fois les données disponibles. Cela dépasse largement les capacités humaines, surtout lorsqu'il s'agit de traiter plusieurs opérateurs simultanément. Nous discuterons de ce point plus en détail lors de la prochaine réunion, car d'autres fonctionnalités doivent encore être ajoutées avant une utilisation en production. Cependant, il est évident que cette automatisation offre un avantage considérable.
- **Diminution d'erreur** : Comme expliqué précédemment, la création de factures nécessite une grande rigueur, notamment en ce qui concerne les dates, l'ordre des actions, et l'utilisation des bonnes données. Grâce à cette automatisation, ces tâches sont désormais moins sujettes aux erreurs humaines, garantissant ainsi une plus grande précision et fiabilité.

8 | Discussions

8.1 Gestion de projet

8.1.1 Déroulement du projet

J'ai commis une erreur d'organisation en débutant mon projet. Au départ, j'étais en parallèle entre l'acquisition de nouvelles connaissances et le développement de l'application. Ce n'est qu'à la fin du sprint 1 que j'ai découvert la Clean Architecture. Initialement, je m'orientais vers une architecture monolithique classique, guidé par la documentation de Microsoft, qui présente les concepts de manière progressive.

Ce changement de cap a eu un impact sur la gestion de mon projet. Heureusement, le sprint 1 (voir l'annexe I.8) était principalement consacré à la mise en place des environnements de développement pour les deux applications, ainsi qu'à l'intégration d'un pipeline CI/CD. Peu de codes avaient été écrits à ce stade, ce qui a facilité la migration vers la Clean Architecture, et j'ai même pu réutiliser certains éléments de codes.

Le sprint 2 (voir l'annexe I.9) s'est concentré sur l'implémentation de la logique métier de l'application, avec l'objectif d'intégrer également l'automatisation. Cependant, cette dernière tâche n'a pas pu être finalisée et a été reportée au sprint 3.

Durant le sprint 3, (voir l'annexe I.10) j'ai réussi à automatiser la génération des factures, à mettre à jour le pipeline CI/CD et à ajouter quelques tests unitaires. Néanmoins, je n'étais pas entièrement satisfait de la couverture de ces tests, principalement en raison du manque de temps. Après la conclusion du sprint 3, j'ai pris le temps de nettoyer le code et d'ajouter des tests supplémentaires pour les méthodes les plus critiques, afin d'assurer une meilleure fiabilité de l'application.

8.1.2 Apprentissage

Ce problème aurait pu être évité si j'avais pris davantage de temps pour explorer et comparer les différentes architectures disponibles pour les APIs. Cependant, grâce à l'utilisation du framework SCRUM, j'ai pu réagir rapidement et adapter le développement en conséquence, ce qui illustre bien l'agilité et la flexibilité que SCRUM apporte au projet.

8.1.3 Avis

Le choix d'utiliser Azure DevOps n'a pas pleinement optimisé la gestion de projet avec SCRUM dans mon cas. Bien qu'il s'agisse d'un outil professionnel offrant de nombreuses fonctionnalités, il nécessite un suivi rigoureux pour en tirer tous les bénéfices.

Dans mon projet, j'aurais pu me contenter d'un simple fichier Excel pour gérer mes user stories, et cela m'aurait probablement permis d'atteindre le même niveau d'avancement en termes de développement. Cependant, l'avantage d'un outil comme Azure DevOps réside dans sa configuration prête à l'emploi, ce qui m'a permis de me concentrer rapidement sur le contenu plutôt que sur le choix et la mise en place du support.

Une amélioration que j'aurais pu apporter durant le développement aurait été de lier mes commits aux user stories correspondantes. Cela aurait facilité le suivi de l'avancement de chaque user story et offert une meilleure traçabilité du projet.

8.2 Mappage

Je dois admettre que l'état de l'application avant le début de ce projet, ainsi que la modélisation existante de la base de données, m'ont rapidement permis de trouver une solution efficace pour mapper les données. La solution consiste à associer une personne de référence (un utilisateur) à un opérateur, puis à stocker l'ID du client Invoice Ninja dans la table des opérateurs.

8.2.1 Avantages

Cette solution s'est révélée simple et rapide à mettre en œuvre. En termes de performances, elle est idéale car elle établit une relation directe entre l'ID du client Invoice Ninja et l'opérateur, ce qui élimine le besoin de joindre plusieurs tables pour générer une facture.

8.2.2 Inconvénients

Cependant, cette intégration présente une limitation : elle ne permet pas de conserver un historique des personnes de référence associées à un opérateur. Pour surmonter cette lacune, une solution serait d'ajouter une nouvelle table, telle que "UserReferenceHistory", qui pourrait suivre les changements dans les associations au fil du temps.

8.3 Génération des factures

8.3.1 Etat des lieux

Le service que j'ai développé pour la génération des factures constitue une bonne base. L'implémentation actuelle permet de récupérer correctement les données en fonction des dates et de calculer les totaux pour les services. J'ai également ajouté des conditions supplémentaires pour renforcer la robustesse du code, en intégrant des tests unitaires afin d'assurer la qualité et la fiabilité de l'application.

8.3.2 Axe d'amélioration

Cependant, plusieurs améliorations sont possibles. Ces évolutions dépendent également de la clarification par Cellsmaniak de sa stratégie de facturation.

Un premier axe d'amélioration serait d'intégrer un service de notification pour informer sur l'état des factures, ce qui améliorerait l'expérience utilisateur en offrant une meilleure visibilité sur le processus de facturation.

Un autre point concerne la modélisation des tables "Contracts" et "OperatorInvoices". Lors du développement, le CTO de Veetamine m'avait demandé de ne pas investir trop d'efforts dans l'utilisation des contrats. C'est pourquoi j'ai opté pour une structure de facturation fixe liée aux services. Cependant, à l'avenir, il serait pertinent que la table "Contracts", associée à "ContractType", définisse le prix des services et les quantités spécifiques à chaque client.

Un dernier axe d'amélioration porte sur la configuration générale d'Invoice Ninja, notamment en ce qui concerne la gestion des langues. Par exemple, Invoice Ninja associe les langues à des valeurs entières (par exemple, 64 pour le français). Lorsqu'un POST est effectué pour créer un client, la langue peut être référencée par une chaîne de caractères. Toutefois, si Invoice Ninja est configuré en anglais et que le POST mentionne "français" pour la langue, l'API d'Invoice Ninja ne reconnaît pas cette information et applique une valeur par défaut.

Ce problème se pose également pour les taxes. J'ai tenté de contourner cette limitation en intégrant une méthode qui effectue une requête GET à l'API d'Invoice Ninja en passant le nom de la taxe en paramètre. Cependant, si le nom de la taxe change, cette méthode pourrait ne plus fonctionner correctement, bien que j'aie prévu une valeur par défaut pour pallier ce risque.

```
1
2  var taxRate = await _taxService.GetTaxRateAsync("TVA Suisses") ?? new TaxRateDto
3      {
4          Name = "TVA Suisses",
5          Rate = 8.1m
6      };
7
```

8.4 Retour d'expérience Clean Architecture

8.4.1 Mise en place

L'utilisation de l'IDE Rider de JetBrains m'a grandement facilité la mise en place des différents projets et couches dans la Clean Architecture. Rider offre des raccourcis et des intégrations spécifiques au framework ASP.NET Core dès qu'on fait un clic droit sur un répertoire, ce qui accélère le processus de développement.

L'aspect visuel de Rider améliore également la compréhension du projet. Contrairement à VS Code, où tous les dossiers restent gris quels que soient leurs rôles, Rider utilise des icônes colorées pour différencier les types de répertoires. Par exemple, un dossier avec une icône violette représente un fichier de solution avec des configurations ".sln", tandis qu'un dossier avec

une icône verte contient le code source qui sera compilé et exécuté. En tant que débutant, ces petites différences visuelles m'ont aidé à mieux comprendre la structure et le fonctionnement du projet.

Concernant la mise en place, j'ai dû m'y prendre à plusieurs reprises avant de trouver une source YouTube.

8.4.2 Structure de l'intégration d'une méthode

Prenons un exemple concret pour illustrer les fichiers utilisés lors de la réalisation d'une requête POST pour ajouter un utilisateur dans mon API :

- **IRepository** : Interface définissant un contrat générique pour l'accès aux données. Elle spécifie notamment la méthode asynchrone "Task AddAsync(T entity) ;". Cette interface appartient à la couche Domain.
- **Repository** : Implémentation de l'interface 'IRepository', incluant la méthode "AddAsync(T entity)". Cette classe, située dans la couche Infrastructure, interagit directement avec la base de données.
- **User Entity** : Classe représentant l'entité "User", avec des propriétés telles que "Email", "PhoneNumber", etc. Cette classe encapsule les règles spécifiques à un utilisateur et fait partie de la couche Domain.
- **IUserService** : Interface spécifique pour la logique métier de l'utilisateur. Il définit la méthode asynchrone "Task AddUserAsync(User entity) ;". Cette interface est située dans la couche Application.
- **UserService** : Implémentation de l'interface "IUserService", responsable de la logique métier pour l'ajout d'un utilisateur. Ce service utilise "IRepository" pour interagir avec la base de données. Il est implémenté dans la couche Application.
- **UserDto** : Le Data Transfer Object (DTO) utilisé pour transférer les données entre les couches. Il sert de modèle de validation pour les données reçues par l'API. Cette classe fait partie de la couche Application.
- **UserController** : Contrôleur qui gère la requête POST "/api/Users". Il reçoit un "UserDto", valide les données, puis appelle "UserService" pour créer l'utilisateur. Cette classe appartient à la couche API.
- **Program.cs** : Point d'entrée de l'application, où les services sont enregistrés dans le conteneur d'injection de dépendances. Ce fichier configure également la base de données et se trouve dans la couche API.
- **ApplicationDbContext** : Classe responsable de la communication avec la base de données via Entity Framework Core. "ApplicationDbContext" gère les configurations des entités et la connexion à la base de données. Elle est située dans la couche Infrastructure.

8.4.3 Avis et recommandations

On peut observer que l'utilisation de cette architecture, bien que puissante et structurée, semble surdimensionnée pour un simple POST d'utilisateur. Aujourd'hui mon application, qui reste relativement simple, atteint déjà un poids de 100 Mo, principalement en raison des nombreuses couches et fichiers de configuration associés à la Clean Architecture.

Cependant, je peux affirmer que cette architecture m'a permis de structurer mon travail de manière plus organisée et méthodique, ce qui sera particulièrement avantageux lorsque le projet gagnera en complexité.

Un autre avantage notable est la réutilisabilité du code. Par exemple, le code du "Repository", qui implémente les opérations CRUD de manière générique, peut être utilisé par plusieurs services. Cela permet d'éviter la duplication du code et de maintenir une cohérence dans les opérations de base. De plus, bien que cette architecture entraîne la création de nombreux fichiers, elle aide à mieux comprendre le code en segmentant les responsabilités, ce qui évite d'avoir des fichiers trop longs et complexes. Cependant, l'inconvénient est que l'on peut rapidement se retrouver avec de nombreux fichiers ouverts simultanément, rendant la navigation difficile. Pour pallier cela, il est essentiel de nommer correctement les fichiers et les méthodes afin de faciliter la recherche et la gestion du code.

Recommanderais-je cette architecture pour Veetamine ? Oui. En tant que développeur, j'ai ressenti un véritable confort en travaillant dans un environnement propre et bien organisé, ce qui est, à mon avis, crucial. Toutefois, cette architecture impose une structure lourde, avec un grand nombre de fichiers, ce qui pourrait ne pas convenir à tous les projets.

Actuellement, Veetamine est une application monolithique, mais il est envisageable qu'elle évolue vers une architecture de microservices. Si ce scénario se réalise, je pense que la Clean Architecture serait particulièrement adaptée. Dans une architecture de microservices, chaque microservice étant dédié à une fonction spécifique, le nombre de fichiers par service serait réduit, facilitant ainsi la gestion et la maintenabilité de la Clean Architecture.

Cependant, il est important de se demander si chaque microservice doit systématiquement adopter une telle architecture. Pour des microservices plus simples, cette complexité pourrait être superflue. Ce point mérite une discussion approfondie avec le CTO de Veetamine et l'équipe de développement afin de déterminer la meilleure approche en fonction des besoins spécifiques de chaque service.

9 | Conclusion

Ce projet avait pour objectif l'intégration d'Invoice Ninja, un service tiers, à l'API de Veetamine, dans le but d'automatiser la facturation des clients de Cellsmaniak. Cette tâche a nécessité une compréhension approfondie du contexte existant. L'analyse préliminaire a révélé la nécessité de redéfinir certains objectifs, notamment en décidant de ne pas imposer une synchronisation forcée, afin de garantir une implémentation réaliste et efficace.

Le résultat de cette intégration est la génération automatique de factures prêtes à être envoyées pour chaque client, en utilisant les données issues de la base de données de Veetamine. Ce processus est programmé pour s'exécuter chaque premier jour du mois à 2 heures du matin.

Pour atteindre ces objectifs, j'ai dû me familiariser avec le framework ASP.NET Core, explorer différentes architectures comme l'architecture N-Tier et la Clean Architecture, puis faire un choix éclairé pour l'implémentation. J'ai également approfondi ma compréhension de l'API Invoice Ninja et trouvé des solutions pour mapper efficacement les données entre les deux API. De plus, j'ai mené des recherches supplémentaires pour choisir un service cloud, conteneuriser l'application, et mettre en place un pipeline CI/CD.

Bien que cette intégration en soit encore à ses débuts, les avantages sont déjà perceptibles, notamment en termes de gain de temps, de centralisation des données et de réduction des erreurs humaines. Une prochaine étape intéressante serait de rendre la facturation plus modulaire en fonction des spécificités de chaque client, car actuellement, chaque client est facturé sur la base de quatre services standardisés.

Ce projet m'a poussé à analyser des solutions existantes, à prendre des décisions et à les implémenter de manière autonome. Bien que cet exercice ne soit pas nouveau pour moi, la différence réside dans le fait que, cette fois-ci, j'étais seul à prendre les décisions. Je suis fier du résultat obtenu et de l'expérience que j'ai acquise tout au long de ce travail.

I | Annexes

I.1 Méthode GenerateInvoicesAsync

```
1
2 public async Task GenerateInvoicesAsync()
3 {
4     _logger.LogInformation("Starting invoice generation.");
5
6     // Retrieve the necessary dates and years/months
7     var (currentDate, currentYear, currentMonth, previousMonth, previousYear,
8         ↪ twoMonthsBefore, twoMonthsBeforeYear) = CalculateDates();
9
10    // Use currentYear and currentMonth
11    var operatorIds = await
12        ↪ _operatorService.GetOperatorsWithInvoicesForLastTwoMonthsAsync(currentYear,
13        ↪ currentMonth);
14    _logger.LogInformation($"Operators with invoices in the last two months found:
15        ↪ {string.Join(", ", operatorIds)}");
16
17    foreach (var operatorId in operatorIds)
18    {
19        await ProcessOperatorInvoices(operatorId, previousMonth, previousYear,
20            ↪ twoMonthsBefore, twoMonthsBeforeYear, currentDate);
21    }
22
23    _logger.LogInformation("Finished invoice generation job.");
24 }
```

I.2 Méthode CalculateDates

```
1
2 internal (DateTime currentDate, int currentYear, int currentMonth, int previousMonth,
3     ↪ int previousYear, int twoMonthsBefore, int twoMonthsBeforeYear)
4     ↪ CalculateDates(DateTime? specificDate = null)
5 {
6     var currentDate = specificDate ?? DateTime.Now;
7     var currentYear = currentDate.Year;
8     var currentMonth = currentDate.Month;
9
10    var previousMonth = currentMonth - 1;
11    var previousYear = currentYear;
12
13    if (previousMonth <= 0)
14    {
15        previousMonth += 12;
16        previousYear -= 1;
17    }
18
19    var twoMonthsBefore = previousMonth - 1;
20    var twoMonthsBeforeYear = previousYear;
```

```
19
20     if (twoMonthsBefore <= 0)
21     {
22         twoMonthsBefore += 12;
23         twoMonthsBeforeYear -= 1;
24     }
25
26     return (currentDate, currentYear, currentMonth, previousMonth, previousYear,
27           ↪ twoMonthsBefore, twoMonthsBeforeYear);
28 }
```

I.3 Méthode ProcessOperatorInvoices

```
1
2     private async Task ProcessOperatorInvoices(int operatorId, int previousMonth, int
3     ↪ previousYear, int twoMonthsBefore, int twoMonthsBeforeYear, DateTime currentDate)
4     {
5         _logger.LogInformation($"Processing invoices for OperatorId: {operatorId}");
6
7         var previousMonthInvoices = await
8         ↪ _operatorInvoiceService.GetInvoicesByMonthAsync(operatorId, previousYear,
9         ↪ previousMonth);
10        var twoMonthsBeforeInvoices = await
11        ↪ _operatorInvoiceService.GetInvoicesByMonthAsync(operatorId, twoMonthsBeforeYear,
12        ↪ twoMonthsBefore);
13
14        if (previousMonthInvoices.Count == 1 && twoMonthsBeforeInvoices.Count == 2)
15        {
16            var previousMonthInvoice = previousMonthInvoices[0];
17            var firstTwoMonthsBeforeInvoice = twoMonthsBeforeInvoices[0];
18            var secondTwoMonthsBeforeInvoice = twoMonthsBeforeInvoices[1];
19
20            if (previousMonthInvoice.IsInvoiceSent)
21            {
22                _logger.LogInformation($"Invoice for OperatorId: {operatorId} for the month
23                ↪ {previousMonth}/{previousYear} already sent. Skipping.");
24                return;
25            }
26
27            var calculatedValues = CalculateInvoiceValues(previousMonthInvoice,
28            ↪ firstTwoMonthsBeforeInvoice, secondTwoMonthsBeforeInvoice);
29
30            var invoiceDto = await PrepareInvoiceDto(operatorId, calculatedValues,
31            ↪ currentDate);
32            await SendInvoice(previousMonthInvoice, invoiceDto);
33        }
34        else
35        {
36            _logger.LogWarning($"Insufficient data to calculate differences for OperatorId:
37            ↪ {operatorId}. Expected 1 invoice from the previous month and 2 invoices from
38            ↪ two months before.");
39        }
40    }
```

I.4 Méthode CalculateInvoiceValues

```

1
2  internal (decimal calculatedTransactionUsed, int calculatedMachinesCount)
   ↳ CalculateInvoiceValues(OperatorInvoice previousMonthInvoice, OperatorInvoice
   ↳ firstTwoMonthsBeforeInvoice, OperatorInvoice secondTwoMonthsBeforeInvoice)
3  {
4      var transactionUsedDifference = secondTwoMonthsBeforeInvoice.TransactionUsed -
   ↳ firstTwoMonthsBeforeInvoice.TransactionUsed;
5      _logger.LogInformation($"Two months before
   ↳ {secondTwoMonthsBeforeInvoice.TransactionUsed} -
   ↳ {firstTwoMonthsBeforeInvoice.TransactionUsed}");
6
7      var machinesCountDifference = secondTwoMonthsBeforeInvoice.MachinesCount -
   ↳ firstTwoMonthsBeforeInvoice.MachinesCount;
8
9      var calculatedTransactionUsed = previousMonthInvoice.TransactionUsed +
   ↳ transactionUsedDifference;
10     _logger.LogInformation($"Calculated TransactionUsed
   ↳ {previousMonthInvoice.TransactionUsed} - {transactionUsedDifference}");
11
12     var calculatedMachinesCount = previousMonthInvoice.MachinesCount +
   ↳ machinesCountDifference;
13
14     return (calculatedTransactionUsed, calculatedMachinesCount);
15 }
16

```

I.5 Méthode PrepareInvoiceDto

```

1
2  internal async Task<InvoiceNinjaDto> PrepareInvoiceDto(int operatorId, (decimal
   ↳ calculatedTransactionUsed, int calculatedMachinesCount) calculatedValues, DateTime
   ↳ currentDate)
3  {
4      var operatorEntity = await _operatorService.GetOperatorByIdAsync(operatorId);
5      var invoiceNinjaClientId = operatorEntity.InvoiceNinjaClientId;
6
7      if (string.IsNullOrEmpty(invoiceNinjaClientId))
8      {
9          _logger.LogError($"Operator with ID {operatorId} does not have a valid Invoice
   ↳ Ninja client ID.");
10         return null;
11     }
12
13     _logger.LogInformation($"Operator {operatorId} has Invoice Ninja Client ID:
   ↳ {invoiceNinjaClientId}");
14
15     var invoiceNumber = await _settingInvoiceNinjaService.GetNextInvoiceNumberAsync();
16
17     var taxRate = await _taxService.GetTaxRateAsync("TVA Suisses") ?? new TaxRateDto
18     {
19         Name = "TVA Suisses",
20         Rate = 8.1m
21     };
22     _logger.LogInformation($"Retrieved tax rate: {taxRate.Name} - {taxRate.Rate}");

```

```
23
24     return new InvoiceNinjaDto
25     {
26         Client_id = invoiceNinjaClientId,
27         Status_id = "1",
28         Is_deleted = false,
29         Number = invoiceNumber,
30         Date = currentDate.ToString("yyyy-MM-dd"),
31         Due_date = currentDate.AddMonths(1).ToString("yyyy-MM-dd"),
32         Uses_inclusive_taxes = false,
33         Tax_name1 = taxRate.Name,
34         Tax_rate1 = taxRate.Rate,
35         Line_items = GenerateLineItems(calculatedValues.calculatedMachinesCount,
36             ↪ calculatedValues.calculatedTransactionUsed),
37         Entity_type = "invoice",
38         Auto_bill_enabled = false
39     };
40 }
```

I.6 Méthode GenerateLineItems

```
1
2     private List<LineItemDto> GenerateLineItems(int calculatedMachinesCount, decimal
3     ↪ calculatedTransactionUsed)
4     {
5         return
6         [
7             new LineItemDto
8             {
9                 Quantity = calculatedMachinesCount,
10                Cost = 7,
11                Product_key = "Client digital service fee",
12                Product_cost = 0,
13                Notes = "Fix fee per machine per month",
14                Discount = 0,
15                Is_amount_discount = true,
16                Sort_id = 0,
17                Line_total = 7 * calculatedMachinesCount,
18                Tax_amount = 0,
19                Gross_line_total = 7 * calculatedMachinesCount,
20                Type_id = 1,
21                Tax_id = 1,
22            },
23            new LineItemDto
24            {
25                Quantity = calculatedTransactionUsed,
26                Cost = 0.0036m,
27                Product_key = "Client variable digital service fee",
28                Product_cost = 0,
29                Notes = "Variable fee per machine per month",
30                Discount = 0,
31                Is_amount_discount = true,
32                Sort_id = 0,
33                Line_total = 0.0036m * calculatedTransactionUsed,
34                Tax_amount = 0,
```

```

35         Gross_line_total = 0.0036m * calculatedTransactionUsed,
36         Type_id = 1,
37         Tax_id = 1
38     },
39
40     new LineItemDto
41     {
42         Quantity = calculatedMachinesCount,
43         Cost = 1,
44         Product_key = "VEETAMINE fix digital service fee",
45         Product_cost = 0,
46         Notes = "Fix fee per machine per month",
47         Discount = 0,
48         Is_amount_discount = true,
49         Sort_id = 0,
50         Line_total = 1 * calculatedMachinesCount,
51         Tax_amount = 0,
52         Gross_line_total = 1 * calculatedMachinesCount,
53         Type_id = 1,
54         Tax_id = 1
55     },
56
57     new LineItemDto
58     {
59         Quantity = calculatedTransactionUsed,
60         Cost = 0.0024m,
61         Product_key = "VEETAMINE variable digital service fee",
62         Product_cost = 0,
63         Notes = "Variable fee per machine per month",
64         Discount = 0,
65         Is_amount_discount = true,
66         Sort_id = 0,
67         Line_total = 0.0024m * calculatedTransactionUsed,
68         Tax_amount = 0,
69         Gross_line_total = 0.0024m * calculatedTransactionUsed,
70         Type_id = 1,
71         Tax_id = 1
72     }
73 ];
74 }
75

```

I.7 Méthode SendInvoice

```

1  internal async Task SendInvoice(OperatorInvoice previousMonthInvoice, InvoiceNinjaDto
   ↪ invoiceDto)
2  {
3      if (invoiceDto == null) return;
4
5      _logger.LogInformation($"Sending invoice to Invoice Ninja with details:");
6      var invoiceResponse = await
   ↪ _invoiceNinjaClientService.CreateInvoiceAsync(invoiceDto);
7
8      if (invoiceResponse.IsSuccess)
9      {
10         previousMonthInvoice.IsInvoiceSent = true;
11         await _operatorInvoiceService.UpdateInvoiceAsync(previousMonthInvoice);

```

```
12     }  
13   }  
14
```

I.8 Sprint 1

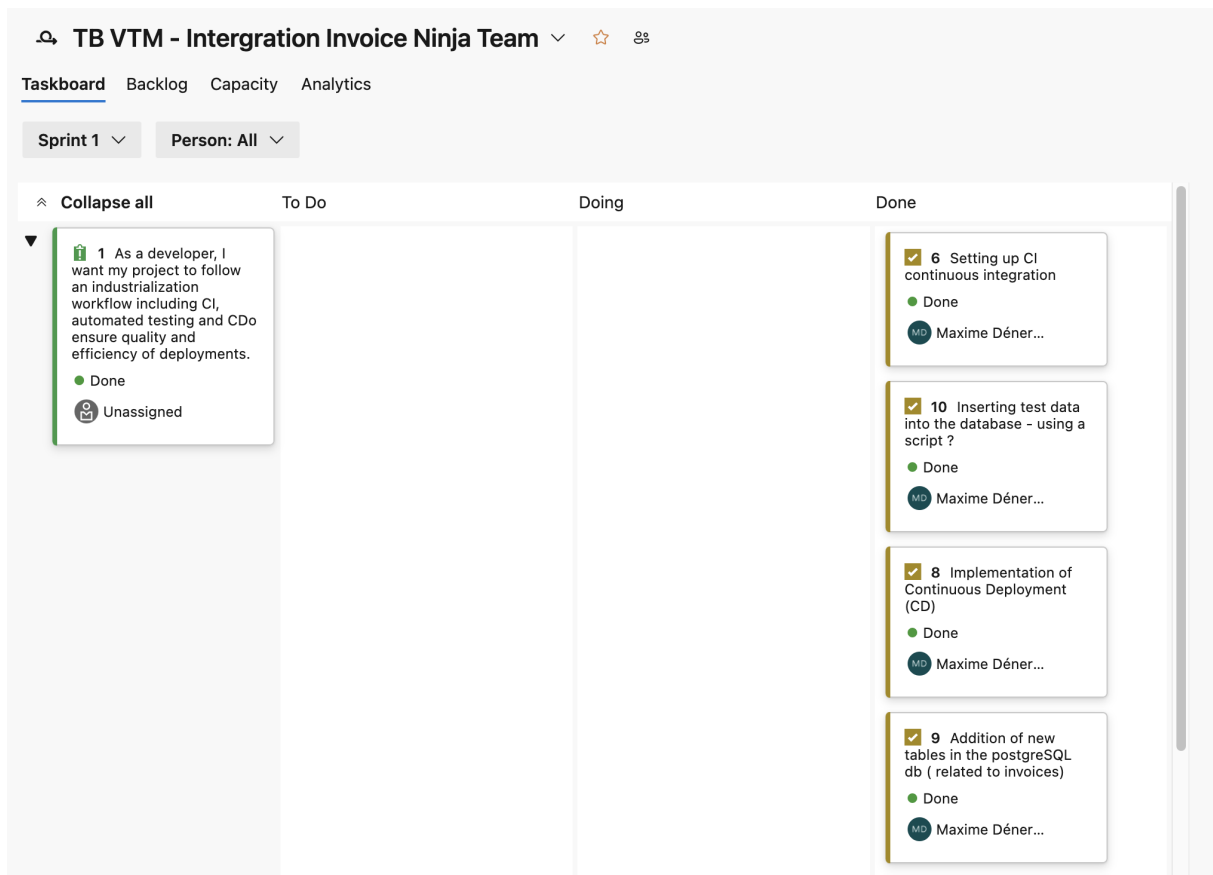


FIGURE I.1 – *Sprint 1 - Azure DevOps*
Source: de l'auteur

I.9 Sprint 2

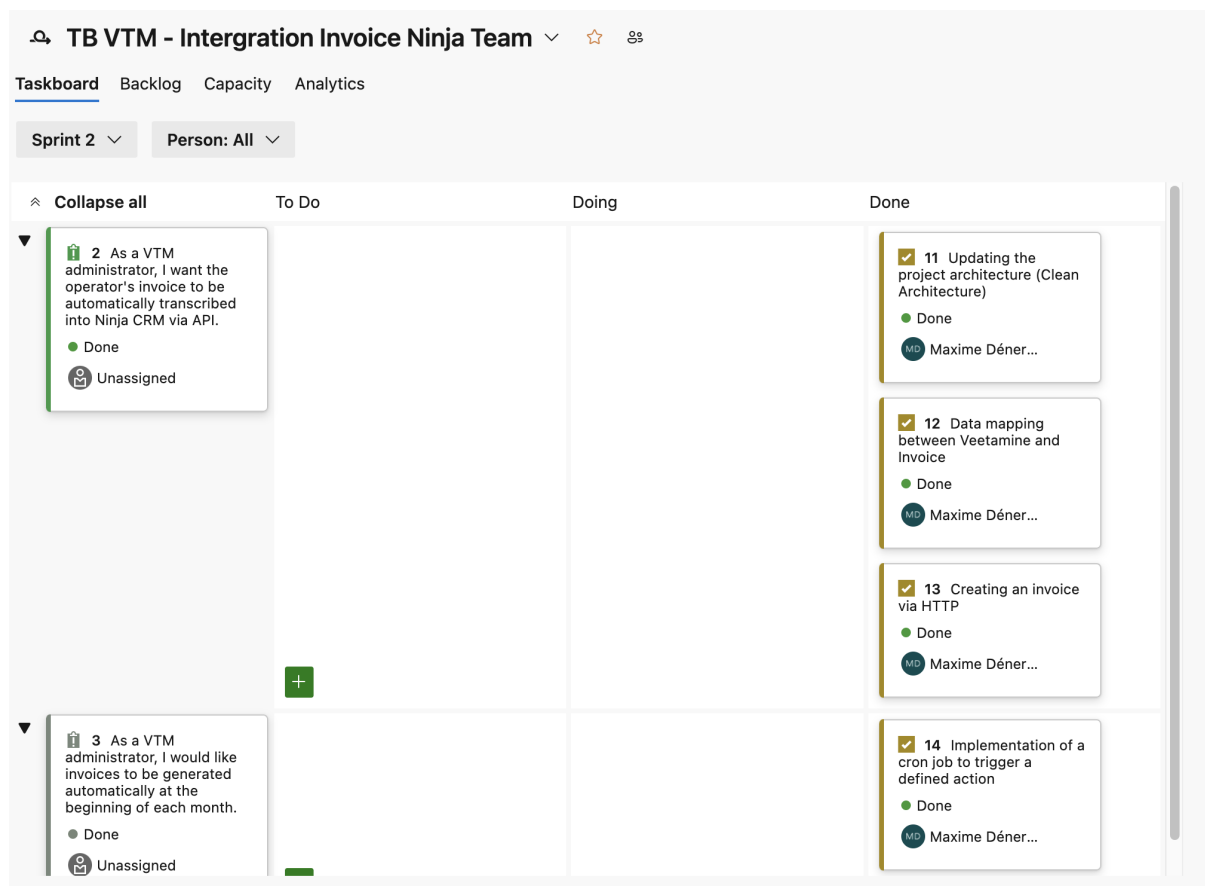


FIGURE I.2 – *Sprint 1 - Azure DevOps*
Source: de l'auteur

I.10 Sprint 3

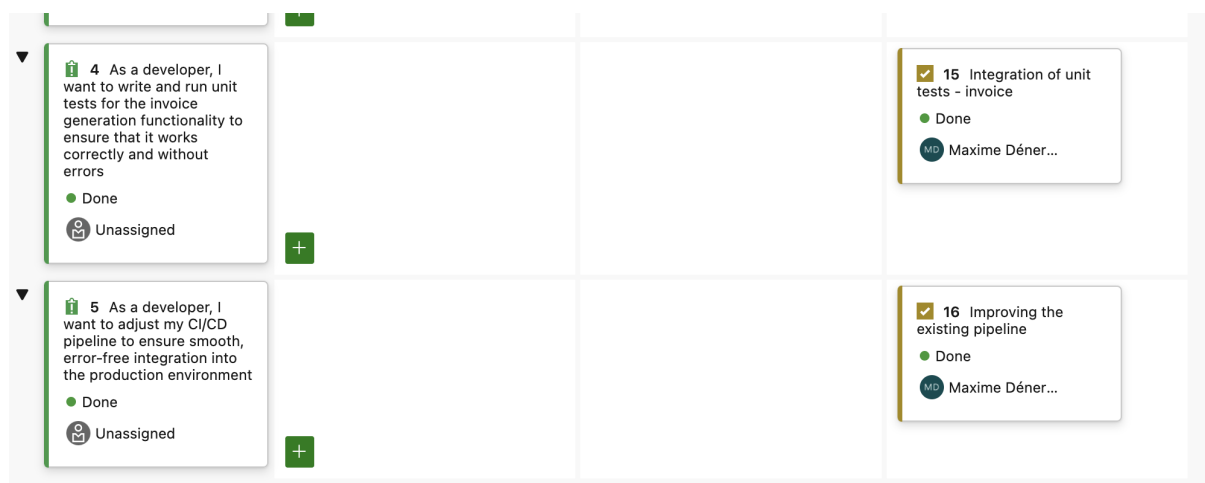


FIGURE I.3 – *Sprint 1 - Azure DevOps*
Source: de l'auteur

Références

- ARK, T. (2024). LE PARTENAIRE PRIVILÉGIÉ DES CRÉATEURS D'ENTREPRISE. Récupérée le à partir de <https://www.theark.ch/fr/page/le-partenaire-privilegie-des-createurs-d-entreprise-1692>
- ATLASSIAN. (2024). Qu'est-ce que Git ? Récupérée le à partir de <https://www.atlassian.com/fr/git/tutorials/what-is-git>
- AWS. (s. d.). Qu'est-ce que Flutter ? Récupérée le à partir de <https://aws.amazon.com/fr/what-is/flutter/>
- AWS. (2024). Quelle est la différence entre MySQL et PostgreSQL ? Récupérée le à partir de <https://aws.amazon.com/fr/compare/the-difference-between-mysql-vs-postgresql/#:~:text=Summary%20of%20differences%3A%20PostgreSQL%20vs%20MySQL,-Category&text=MySQL%20has%20limited%20support%20of%20stored%20procedures%20in%20multiple%20languages.>
- CLOUD, G. (2024). Quelles sont les principales différences entre PostgreSQL et SQL Server ? Récupérée le à partir de <https://cloud.google.com/learn/postgresql-vs-sql>
- CLOUDFLARE. (2024). Qu'est-ce que l'OWASP ? Qu'est-ce que le top 10 de l'OWASP ? Récupérée le à partir de <https://www.cloudflare.com/fr-fr/learning/security/threats/owasp-top-10/>
- DATASCIENTEST. (2020). Docker : qu'est-ce que c'est et comment l'utiliser ? Récupérée le à partir de <https://datascientest.com/docker-guide-complet>
- for GEEKS, G. (2024). React Components. Récupérée le à partir de <https://www.geeksforgeeks.org/reactjs-components/>
- G2. (2024). Compare Render and Salesforce Heroku. Récupérée le à partir de <https://www.g2.com/compare/render-render-vs-salesforce-heroku>
- GADGE, A. (2019). Is it worth deploying Database or Data Store on Containers ? Récupérée le à partir de <https://www.ashnik.com/is-it-worth-deploying-database-or-data-store-on-containers/>
- GIESEL, S. (2023). How to write your own cron Job scheduler in ASP.NET Core. Récupérée le à partir de <https://steven-giesel.com/blogPost/fb1ce2ab-dd27-43ed-aaab-077adf2d15cd>
- GRAFIKART. (2019). Documenter son API avec OpenAPI (Swagger). Récupérée le à partir de <https://grafikart.fr/tutoriels/swagger-openapi-php-1160>

- HEDDINGS, A. (2020). Should You Run a Database in Docker? Récupérée le à partir de <https://www.howtogeek.com/devops/should-you-run-a-database-in-docker/>
- HEROKU. (2024). Heroku Postgres. Récupérée le à partir de <https://elements.heroku.com/addons/heroku-postgresql>
- INZA, J. M. (2024). .NET : faisons le point. In *Dossier spécial .NET* (p. 38-41).
- KINSTA. (2023). Qu'est-ce que TypeScript? Un guide complet. Récupérée le à partir de <https://kinsta.com/fr/base-de-connaissances/guide-complet-typescript/>
- KUMAR, A. (2023). Choosing Between Controllers and Minimal API for .NET APIs. Récupérée le à partir de <https://www.c-sharpcorner.com/article/choosing-between-controllers-and-minimal-api-for-net-apis/>
- MICROSOFT. (2023). Rate limiting middleware in ASP.NET Core. Récupérée le à partir de <https://learn.microsoft.com/en-us/aspnet/core/performance/rate-limit?view=aspnetcore-8.0>
- MICROSOFT, L. (2023a). Architecturer des applications web modernes avec ASP.NET Core et Azure - Architectures courantes des applications web. Récupérée le à partir de <https://learn.microsoft.com/fr-fr/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures>
- MICROSOFT, L. (2023b). Architecturer des applications web modernes avec ASP.NET Core et Azure - Principes de l'architecture. Récupérée le à partir de <https://learn.microsoft.com/fr-fr/dotnet/architecture/modern-web-apps-azure/architectural-principles>
- MICROSOFT, L. (2023c). Entity Framework Core. Récupérée le à partir de <https://learn.microsoft.com/fr-fr/ef/core/>
- MICROSOFT, L. (2023d). Utilisation de données dans les applications ASP.NET Core. Récupérée le à partir de <https://learn.microsoft.com/fr-fr/dotnet/architecture/modern-web-apps-azure/work-with-data-in-asp-net-core-apps>
- MICROSOFT, L. (2023e). Vue d'ensemble des migrations. Récupérée le à partir de <https://learn.microsoft.com/fr-fr/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli>
- MICROSOFT, L. (2024). Stratégie de langage Microsoft .NET. Récupérée le à partir de <https://learn.microsoft.com/fr-fr/dotnet/fundamentals/languages>
- MOZILLA. (s. d.). WebSockets. Récupérée le à partir de https://developer.mozilla.org/fr/docs/Web/API/WebSockets_API

Références

- NINJA, I. (2024a). Docker for Invoice Ninja. Récupérée le à partir de <https://github.com/invoiceninja/dockerfiles>
- NINJA, I. (2024b). Notre histoire. Récupérée le à partir de <https://invoiceninja.com/our-story/>
- NINJA, I. (2024c). Ressources pour développeurs. Récupérée le à partir de https://invoiceninja.github.io/fr_CA/developer-guide/
- NINJA, I. (2024d). Self-Hosting Invoice Ninja. Récupérée le à partir de <https://www.invoiceninja.org/>
- ODILE, G. (2022). Les bases de la clean architecture. Récupérée le à partir de <https://www.ssw.com.au/rules/rules-to-better-clean-architecture/>
- OLESON, T. (2023). Aggregates and Architecture. Récupérée le à partir de <https://www.linkedin.com/pulse/aggregates-architecture-tim-oleson/>
- ORACLE. (2024). Pourquoi le PaaS est-il important ? Récupérée le à partir de <https://www.oracle.com/ch-fr/cloud/what-is-paas/benefits-of-paas/>
- OWASP. (2019). API4 :2019 Lack of Resources and Rate Limiting. Récupérée le à partir de <https://owasp.org/API-Security/editions/2019/en/0xa4-lack-of-resources-and-rate-limiting/>
- PERRAUDEAU, A. (2021). Architecture Monolithique. Récupérée le à partir de <https://aperrauveau.medium.com/architecture-monolithique-abc2ae636b51>
- PORTO, P. (2018). 4 branching workflows for Git. Récupérée le à partir de <https://medium.com/@patrickporto/4-branching-workflows-for-git-30d0aaee7bf>
- POUCHELET, F. (2024). Le guide du succès du développeur. In *Dossier spécial .NET* (p. 4-7).
- QUARTZ.NET. (2023). Quartznet. Récupérée le à partir de https://github.com/quartznet/quartznet/blob/main/database/tables/tables_postgres.sql
- QUARTZ.NET. (2024). Microsoft Dependency Injection Integration. Récupérée le à partir de <https://www.quartz-scheduler.net/documentation/quartz-3.x/packages/microsoft-di-integration.html#persistent-job-stores>
- RADIX. (2024). Vite JS – Next Generation Frontend Tooling. Récupérée le à partir de <https://radixweb.com/introduction-of-vite-js>
- RENDER. (2024). Predictable pricing that scales with you. Récupérée le à partir de <https://render.com/pricing>

- ROUSSEZ, L. (2017). Qu'est-ce que Docker ? Quelles sont les limites de la technologie ? Ses évolutions et son adaptation aux besoins des environnements de production. Récupérée le à partir de <https://www.linkedin.com/pulse/quest-ce-que-docker-queelles-sont-les-limites-de-la-ses-roussez/>
- RULES, S. (2024). Rules to Better Clean Architecture. Récupérée le à partir de <https://www.ssw.com.au/rules/rules-to-better-clean-architecture/>
- SIVA. (2023). Minimal APIs in ASP.NET Core : Compare With Controller. Récupérée le à partir de <https://www.c-sharpcorner.com/blogs/minimal-apis-in-asp-net-core-a-lean-approach-to-web-development>
- SVIX. (s. d.). Webhooks vs API Polling. Récupérée le à partir de <https://www.svix.com/resources/faq/webhooks-vs-api-polling/>
- VEETAMINE. (2024a). How It Works. Récupérée le à partir de <https://veetamine.com/en/how-it-works>
- VEETAMINE. (2024b). Partners. Récupérée le à partir de <https://veetamine.com/en/partners>
- WIKIPEDIA. (2024a). GitHub. Récupérée le à partir de <https://en.wikipedia.org/wiki/GitHub>
- WIKIPEDIA. (2024b). React. Récupérée le à partir de <https://fr.wikipedia.org/wiki/React>

Informations sur ce travail

Informations de contact

Auteur : Maxime Dénervaud

HES-SO Valais-Wallis

E-mail : *maxime.denervaud@students.hevs.ch*

Déclaration sur l'honneur

Je déclare, par ce document, que j'ai effectué le travail de bachelor ci-annexé seul, sans autre aide que celles dûment signalées dans les références, et que je n'ai utilisé que les sources expressément mentionnées. Je ne donnerai aucune copie de ce rapport à un tiers sans l'autorisation conjointe du RF et du professeur chargé du suivi du travail de bachelor, à l'exception des personnes qui m'ont fourni les principales informations nécessaires à la rédaction de ce travail.

Lieu, date : Sierre, 16 août 2024

Signature :  _____