Mémoire de fin d'études

Migration de bases de données

Julien Ivars

Master informatique et mobilité
UHA 4.0.5 - Promotion 2024 / 2025

Alternance réalisée chez UNIT SOLUTIONS AG

Tuteur professionnel
M. Cédric Martin

Tuteur pédagogique
M. Mounir ELBAZ

Remerciements

J'aimerais remercier Monsieur le Directeur d'Unit Solutions M. Thierry MOEBEL pour m'avoir donné l'opportunité de rejoindre l'entreprise et d'effectuer ma seconde année de master en alternance. Je le remercie également d'avoir pris en compte mes intérêts en me confiant un projet captivant, correspondant parfaitement aux attentes de mon année. De plus, je suis reconnaissant qu'il ait prolongé mon contrat pour l'année prochaine, me permettant ainsi de me lancer dans le monde du travail et de poursuivre mon évolution au sein de l'entreprise.

Je souhaite exprimer ma gratitude envers M. Cédric MARTIN, mon tuteur en entreprise, pour son accompagnement tout au long de l'année sur le projet. Sa transmission de connaissances techniques et ses explications sur l'architecture et le fonctionnement du projet ont été d'une grande aide pour moi.

Je remercie chaleureusement tous mes collègues chez Unit Solutions pour leur partage de connaissances, leur bonne humeur et leur soutien.

Je tiens à exprimer ma reconnaissance envers toute l'équipe pédagogique de l'UHA 4.0, notamment M. Mounir ELBAZ, M. Pierre-Alain MULLER, M. Florent BOURGEOIS, M. Daniel DA FONSECA, M. Pierre SCHULLER et Mme. Audrey BRUNSPERGER, ainsi que les étudiants de l'UHA 4.0. Leur soutien, leur partage de connaissances, leur accompagnement et leurs conseils au long de l'année m'ont permis de mener à bien mon projet professionnel.

Enfin, je souhaite exprimer ma gratitude envers les relecteurs de ce rapport pour leurs précieux conseils, qui m'ont permis de mener à bien l'écriture de ce rapport.

Sommaire

Introduction

Après avoir réalisé mon parcours de licence professionnelle « Développeur informatique » au sein de l'UHA et obtenu mon diplôme, j'ai souhaité approfondir mes connaissances rejoignant le cursus master proposé par l'UHA 4.0 qui fait suite à la licence.

Mon parcours de master a été réalisé au sein de l'entreprise Unit Solutions basée à Allschwil en Suisse, qui s'était déjà proposée de me suivre dans mon cursus universitaire pour les deux années précédentes. Mes contributions principales se sont orientées sur le projet InfSuite et l'environnement l'entourant. L'application pour laquelle j'ai pu apporter ma contribution a comme objectif premier de gérer le suivi et la maintenance d'état d'ouvrages d'art.

Dans ce mémoire, je vous présenterais les détails du projet InfSuite et de ma contribution au projet. J'ai eu pour objectif principal de planifier et de réaliser une migration de base de données. En effet, la base de données étant un point clef de l'application, une maintenance de cette dernière est nécessaire pour assurer une certaine pérennité de l'application. Cette étape de migration s'inscrit dans un projet de maintenir les technologies de l'application à jour et de permettre de palier à d'autres problèmes.

Dans ce document, je commencerai par présenter ce qui m'a amené à rejoindre le cursus master et les compétences acquises durant ma formation, j'aborderais par la suite les enjeux, une analyse et le plan d'action de la migration et, puis j'expliquerais la réalisation et les problèmes rencontrer et enfin, je pourrais conclure ce document.

1. Contexte

Dans cette première section, je vais remettre en contexte le concept de formation, détailler mon parcours académique et présenter l'entreprise qui m'accueille en alternance.

1.1. Présentation de la formation

L'UHA 4.0 propose des formats de formation légèrement différents des cursus traditionnels pour favoriser une immersion professionnelle tout en poursuivant des études universitaires. La formation inclut un stage obligatoire d'une durée minimale de six mois en complément de la période de formation. Cette immersion professionnelle fournit aux étudiants les outils essentiels pour intégrer le monde du travail. La première année introduit le développement et fournit les outils fondamentaux pour comprendre la logique de la programmation. La deuxième année approfondit ces connaissances en abordant la programmation orientée objet avec des langages tels que Java ou C#. La troisième année de Licence ajoute une méthodologie approfondie de gestion de projet en mode agile.

Après ces trois années, les étudiants peuvent, sur admission, rejoindre le parcours master de l'UHA 4.0. Ce parcours aborde les fondements de la recherche et des thématiques variées telles que la cybersécurité et l'algorithmie. Le master, également en alternance, impose une période en entreprise plus soutenue, passant de 6 à 9 mois, tout en réalisant les projets donnés par les encadrants de l'UHA 4.0.

1.2. Mon parcours académique

1.2.1. Jusqu'à la licence

Après avoir obtenu un baccalauréat scientifique, option science de l'ingénieur, j'ai choisi de m'orienter vers le cursus proposé par l'UHA 4.0. J'ai alors pu réaliser mes trois premières années d'études supérieures en engrangeant de nombreuses connaissances en toute autonomie dans le de l'informatique orienté web et de côtoyer des milieux professionnels via les projets et les stages.

J'ai pu conclure ces trois premières années d'études à l'UHA 4.0 il y a deux ans en obtenant ma licence professionnelle.

1.2.2. Le parcours master

Une année à l'UHA 4.0 du parcours « Master Informatique et Mobilité » se divise en deux parties dupliquées sur deux années. Dans un premier temps, l'étudiant doit réaliser, en groupe de 3 ou 4 élèves, un « fil rouge ». Ce projet a pour but de mettre en pratique les connaissances acquises durant les séances de cours assurées par des enseignants chercheurs ou autres intervenants. Les cours portent sur les technologies qui pourront et seront réutilisées durant le fil-rouge tel que l'Intelligence Artificielle, l'optimisation combinatoire, la fouille de données, etc. La seconde partie de l'année est une phase de neufs moins durant laquelle l'étudiant est dans une entreprise pour mettre en pratique, consolider et acquérir de nouvelles connaissances.

Lors de ma première année du parcours de master, nous avons eu comme sujet de fil rouge, la gestion et l'automatisation de l'arrosage des plantes. Le but était de pouvoir récolter des données environnementales liées à une plante, puis les réutiliser pour prendre des décisions à l'aide de divers outils.
Nous avons pu utiliser une sonde de température et un capteur d'humidité pour récolter les données, les envoyer grâce à un microcontrôleur vers un serveur pour stocker et traiter les données.
Nous avons également pu utiliser des caméras pour nous permettre d'effectuer des constatations d'assèchement à partir des feuilles d'une plante qui réagit de manière assez marquée à ce facteur.
J'ai eu comme principales tâche de mettre en place un système IOT pour récolter les données environnementales et les images puis les envoyer vers un serveur distant. J'ai par ailleurs pu participer aux développements et à la recherche de solutions d'analyse des données collectées.

Cette année, pour la seconde année de mon parcours au sein du master, le sujet était de pouvoir prédire le confort moyen d'une salle de travail basé sur le ressenti des utilisateurs. Basé sur de la collecte de données et de l'analyse de ces dernières pour effectuer des prédictions, nous avions accès à une sonde de température, des microphones et des caméras pour analyser l'environnement. Comme à l'itération précédente du fil-rouge, un système de stockage et d'analyse des données nous a permis de réaliser des prédictions basées sur les données collectées. En tant que Scrum Master, j'ai pu développer un système IOT de collecte de données et mettre en place un système de prédiction par analyse des données entrantes.

Mes deux années de master ont consécutivement été réalisées au sein de l'entreprise Unit Solutions.

1.3. L'entreprise Unit solutions

Unit Solutions est une entreprise suisse, basée à Allschwil dans le canton de Bâle-Campagne. Fondée en 1986 premièrement sous le nom CADRZ, dédiée à la création d'un cadastre numérique pour la ville d'Allschwil, elle est aujourd'hui dirigée par M. Thierry MOEBEL.
La politique de développements de l'entreprise va par la suite changer pour finalement réaliser ses propres logiciels et en faire la maintenance. L'entreprise compte actuellement une vingtaine d'employés pour le développement des différentes solutions, le support et l'administratif.

Les développements au sein de l'entreprise reposent sur quatre gros projets:

- Langsam Verkehr2 est une solution visant la gestion des sentiers de mobilité douce en Suisse. Cela comprend la gestion des sentiers de randonnée ou les pistes cyclables.
- Kuba et InfKuba sont les deux solutions principales portées par Unit Solutions. Ces deux logiciels sont relativement similaires dans leur but, mais leur conception est totalement différente. Kuba est sous forme de client lourd (application à installer) tandis qu'InfKuba est une application web qui ne nécessite rien d'autre qu'un navigateur. Elle est la version moderne de Kuba.
- Observo est un projet servant d'extension au projet InfKuba, ou alors d'outil complètement indépendant pour gérer des domaines d'applications qui lui sont propres. Il permet par exemple de gérer de petits ouvrages, du mobilier urbain…

Durant mon alternance, j'ai pu intégrer l'équipe chargée des développements pour la suite logicielle InfSuite. Entouré de plusieurs collègues, j'ai pu développer mes bases de connaissances sur le projet et les différentes technologies utilisées.

2. Etat de l'art

2.1. Présentation d'InfSuite

Début **2016**, les différents cantons suisses utilisaient le client lourd Kuba, l'application mère d'InfSuite. Ce logiciel était payé par l'OFROU pour une meilleure gestion des infrastructures routières en Suisse.

En **2019**, elle lance un appel d'offres dans le but de réduire les coûts liés aux applications qu'elle utilise. L'application Kuba étant une application hautement complète et complexe, le budget n'était alors pas forcément adapté à tous les cas d'usages des différents cantons, certains ayant des besoins différents en termes de gestion.

Unit Solutions a alors décidé de se lancer dans la conception d'une nouvelle solution nommée InfSuite, qui reprend le principe de l'application mère Kuba. La grande différence sur cette nouvelle application est le découpage plus fin des différentes fonctionnalités. Elle permet de proposer aux différents clients de ne leur octroyer l'accès à certaines fonctionnalités que s'ils en ont réellement besoin et permet ainsi de faire coller le budget au besoin.

Les principaux objectifs de l'application étaient donc :

- De n'implémenter que les fonctionnalités que les cantons seraient susceptibles d'utiliser.
- Vendre l'application à l'OFROU grâce à la refonte de l'application qui réduit les coûts de l'application.
- Livrer une version utilisable au bout d'un an et demi maximum pour permettre aux cantons de tester l'application et de s'adapter au nouveau système.

Finalement adoptée par la suite par la majorité des cantons suisses, l'application InfSuite est une application géographique basée sur des technologies web récentes. Le but de l'application est de permettre de faire l'état et le suivi d'ouvrages d'art dans le temps. Lorsqu'un organisme responsable de la préservation des ouvrages d'art vient faire des constatations à une certaine date d'un ouvrage, il enregistre toutes les informations et données relatives dans l'application. Lorsqu'une expertise ultérieure est réalisée sur l'ouvrage, de nouvelles observations seront ajoutées et avec ces nouvelles données l'application fournira un résumé de l'évolution de cet ouvrage, s'il a un comportement particulier, s'il faut planifier une intervention de conservation, etc.

Toutes ces données sont accessibles depuis n'importe quel terminal et depuis n'importe où tant que le client possède une connexion internet et un compte ayant les droits

requis pour accéder aux données. La suite logicielle InfSuite comprend plusieurs types d'outils, tous basés sur le même fonctionnement, mais avec des domaines d'application différents pour permettre d'effectuer des constatations plus spécifiques en fonction du domaine ciblé. Par exemple, sur l'outil InfAqua, il est possible de faire l'observation de cours d'eau (lit de rivières, berges…), InfRail des voies ferrées, InfVias de routes… Pour conserver la compréhensibilité de ce document, InfKuba sera l'outil de référence pour le fonctionnement technique de l'application.

L'application présente de manière simplifiée les données relatives aux différentes constatations sur une carte. Elle représente sous forme de pastilles colorées la note moyenne des ouvrages par zones géographiques lorsque plusieurs ouvrages se trouvent très proches les uns des autres comme le montre l'image Fig. 1.

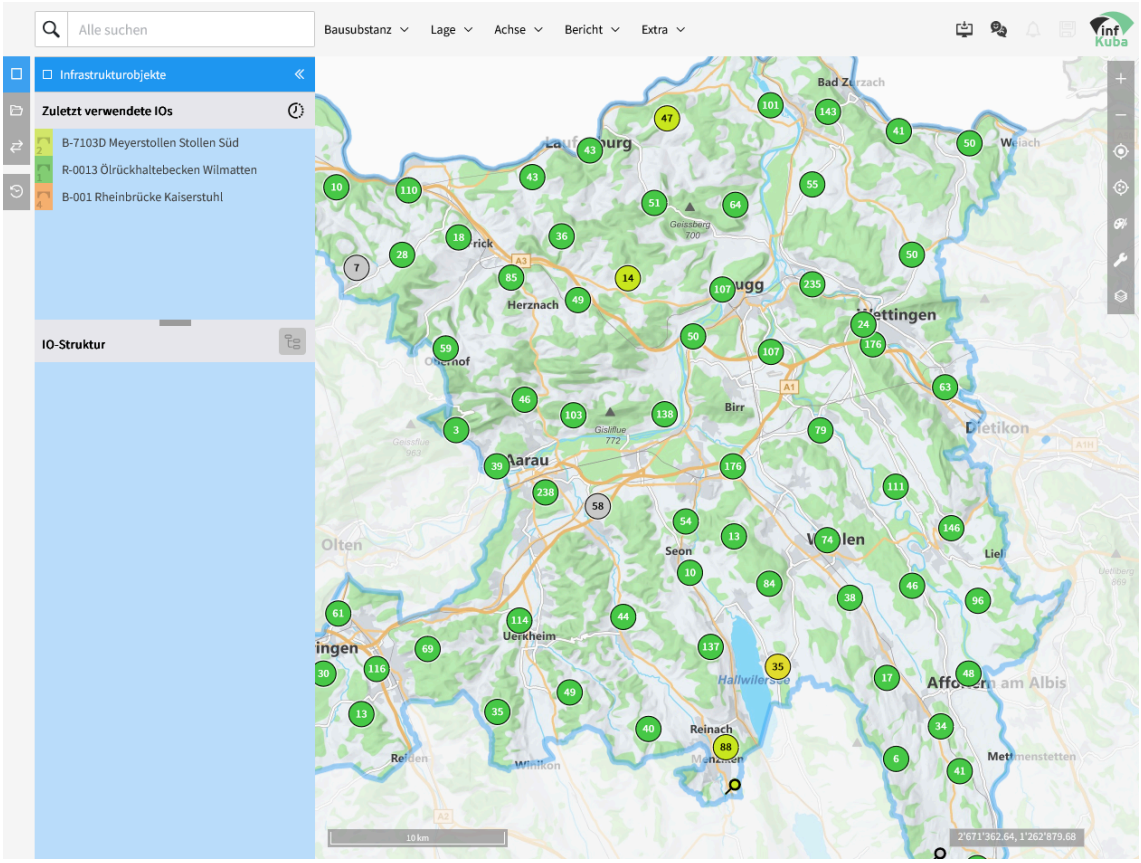

Fig. 1. – Interface principale de l'application InfKuba

En fonction du niveau de zoom, les différents ouvrages se séparent les uns des autres et permettent de voir la dernière note calculée pour l'ouvrage sous forme d'une petite épingle qui représente l'objet d'infrastructure. La couleur varie du vert au rouge, indiquant respectivement que l'ouvrage est en bon état ou qu'il faut intervenir le plus rapidement possible, comme le montre l'image Fig. 2.

En ouvrant les détails de l'objet, on peut retrouver un onglet regroupant les informations relatives à l'ouvrage d'art sous le même format que présenté Fig. 3. On y retrouve des données basiques comme son nom, le canton propriétaire de l'ouvrage, ses dimensions, des commentaires, sa position géographique, des photos, etc.

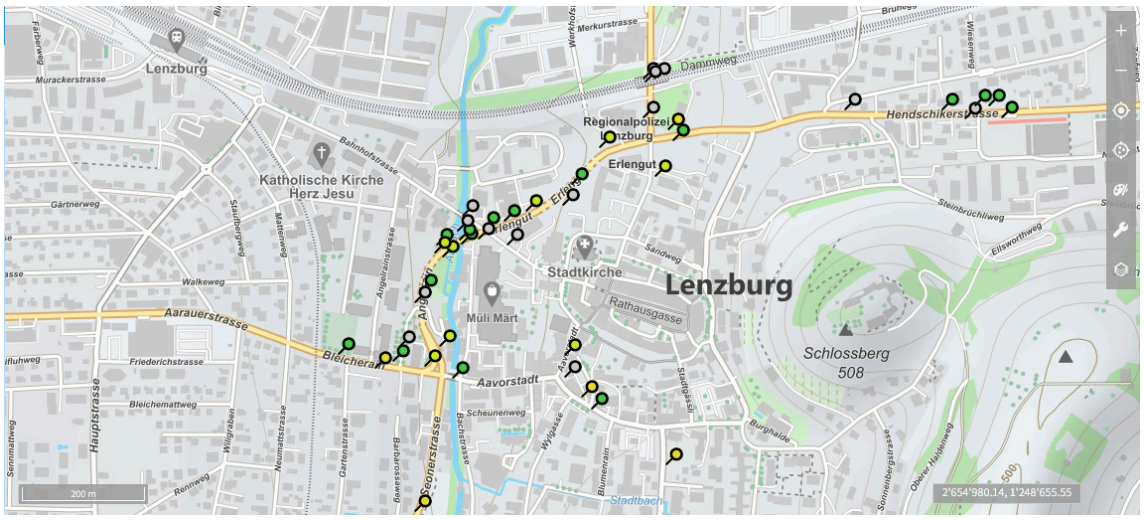

Fig. 2. – Affichage individuel des ouvrages sur la carte

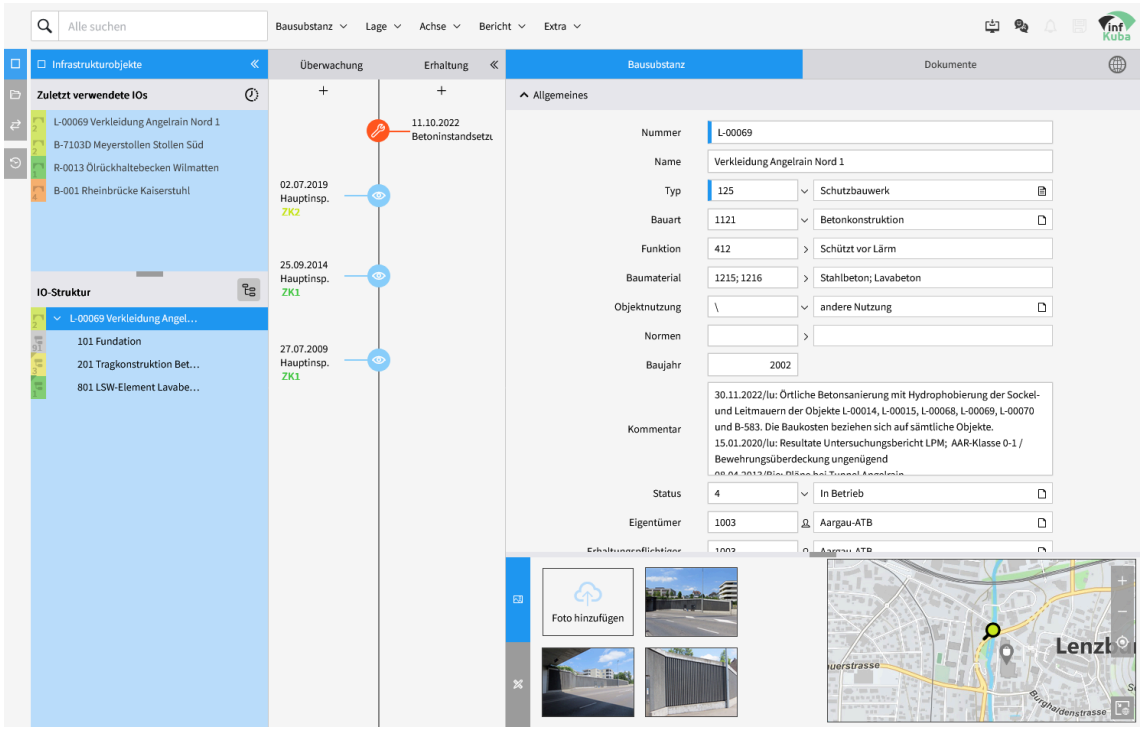

Fig. 3. – Affichage des informations détaillées d'un ouvrage dans l'application

D'autres onglets permettent de retrouver des informations plus précises telles que les observations de l'ouvrage, des graphiques sur l'évolution de l'ouvrage dans le temps, visualiser l'ouvrage en 3D (si fourni par le client)…

Chaque ouvrage appartient à un canton, mais il peut être prêté à d'autres cantons, comme dans le cas où un pont serait partagé entre deux cantons par exemple. Un autre exemple d'ouvrage partagé est lorsque la gestion est déléguée à une ville plutôt qu'au canton. À ce moment, dans l'application, l'ouvrage appartient au canton et est « prêté » à la ville qui en est chargée. Chaque donnée générée à partir de l'application est rattachée au client qui en est à l'origine, ce qui permet d'avoir un historique en cas de problème. Pour permettre de rendre l'application plus légère, l'application est basée sur une architecture trois tiers.

Le premier tiers de l'application correspond à la base de données. Cette base de données est une base PostgreSQL. C'est ici que toutes les données sont stockées ainsi que les relations entre chaque donnée existante. Je détaillerai les particularités techniques de cette base de données qui peuvent expliquer la pertinence de son utilisation dans ce contexte plus loin dans ce rapport.

Le second tiers de l'application correspond à l'API. Une API web est une interface de programmation qui permet à des applications informatiques de communiquer entre elles via Internet. Elle définit les règles et les formats de données pour faciliter l'échange d'informations entre les différentes applications.

Cette API est le serveur back-end qui permet de faire la relation entre le monde extérieur et les données brutes stockées en base de données. Le serveur est développé en C# avec l'ORM Entity Framework. Cette interface permet notamment de mettre en forme les données pour qu'elles correspondent aux besoins du troisième et dernier tiers.

Le dernier tiers correspond à la partie visible de l'application. Appelé frontend, il met à disposition de manière visuelle et simplifiée les données. Ce dernier tiers permet également à l'utilisateur de créer, modifier ou effacer des données. Il est développé en TypeScript avec le framework Angular pour permettre de créer une application dynamique et réactive.

Le dernier service complémentaire est un serveur GIS. Ce serveur permet de fournir des informations cartographiques comme des adresses. Il permet aussi de délivrer les fonds de cartes. Ils peuvent être hébergés et entretenus par Unit Solutions, ou alors par d'autres entreprises comme le fond de carte Open Street Map appartenant à la fondation du même nom. Il n'est pas intégré dans le concept d'architecture trois tiers puisque, comme je l'ai indiqué, ce service est indépendant et n'est pas une dépendance du reste.

Le schéma Fig. 4 ci-dessous représente les trois tiers de l'application communiquant ensemble. L'application propose à l'utilisateur de personnaliser toute l'application. Cela comprend par exemple quel fond de carte afficher, quelles données afficher, personnaliser l'affichage de ces données tel que le type d'affichage des épingles… Ces

configurations sont propres à l'utilisateur et sont sauvegardées en base de données, en parallèle de toutes les autres données de l'application.

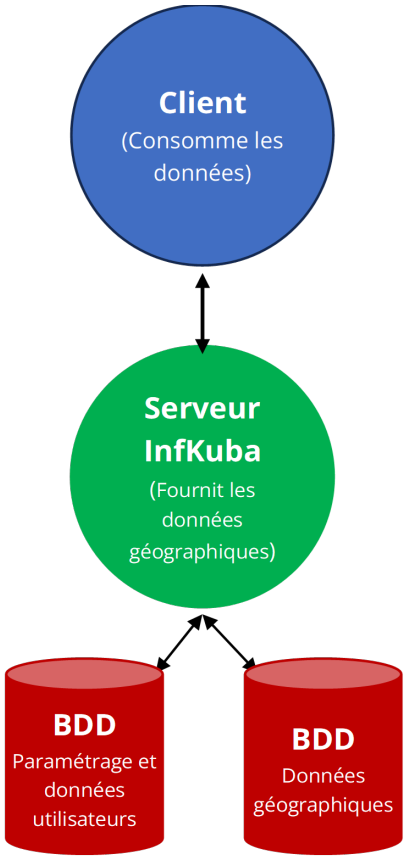

Fig. 4. – Représentation de l'architecture trois tiers d'InfSuite

2.2. PostgreSQL, un système open source

Ce document est axé sur le travail réalisé sur une base de données. L'application utilisant une base de données PostgreSQL, je présente dans cette section ce qu'est PostgreSQL, les avantages de cette technologie, les inconvénients et enfin une conclusion à cette section orientée.

2.2.1. Présentation de PostgreSQL

PostgreSQL est un système de gestion de base de données relationnelle (SGBDR). Le projet est initié en 1986 par Michael Stonebraker et Andrew Yu à l'Université de Californie à Berkeley.

L'une des forces majeures de ce système est d'être Open Source, ce qui signifie qu'il est développé et maintenu par la communauté en plus des développements apportés par la société mère PostgreSQL.

PostgreSQL tient sa réputation de sa fiabilité, sa robustesse et sa richesse fonctionnelle que je détaillerai juste après.

2.2.2. Les principes de base

Comme dit précédemment, PostgreSQL est un SGBDR. Il utilise le langage SQL[1] pour chercher ou manipuler les données stockées. Le système met à disposition une série de fonctions pour permettre ces interactions, à savoir :

- Les transactions : un ensemble d'une ou de plusieurs opérations regroupées en une seule opération atomique.
- Les vues : table virtuelle qui sélectionne et affiche des données à partir d'une ou plusieurs tables réelles.
- Les contraintes d'intégrité : règles qui garantissent la validité et la cohérence des données dans une base de données.
- Les procédures stockées : programme écrit en SQL qui est stocké dans une base de données et peut être exécuté à la demande.
- Les triggers : procédure stockée qui est automatiquement exécutée en réponse à un événement spécifique sur une table.
- Les fonctions utilisateurs : procédure stockée qui renvoie une valeur et peut être utilisée dans une requête SQL comme une fonction intégrée.

PostgreSQL a également l'avantage d'être multiplateforme. Il peut ainsi fonctionner sur des environnements variés avec des systèmes d'exploitation différents, comme Windows, Linux, Mac, etc. L'une des forces de ce système de gestion de base de données

[1] Structured Query Language

réside dans sa capacité à gérer des volumes importants de données allant jusqu'à plusieurs Téraoctets. Cette gestion passe par différents points clés, à savoir :

- L'indexation
- Le partitionnement
- La gestion du cache
- Des notions de concurrence et d'isolation
- De la réplication et du sharding.

2.2.3. Les avantages de PostgreSQL

PostgreSQL est un SGBDR très populaire pour plusieurs raisons :

- Il est open source, ce qui signifie qu'il est gratuit et que son code source est disponible pour tous. Cela permet à la communauté de développeurs de contribuer à son amélioration et de créer des extensions pour ajouter des fonctionnalités supplémentaires.
- Il est très fiable et robuste, ce qui en fait un choix idéal pour les applications critiques et les environnements de production.
- Comme vu précédemment, il est très performant, grâce à son moteur de stockage et son optimiseur de requêtes. Il est capable de gérer de gros volumes de données et de supporter des charges de travail élevées.
- Le système au complet est très flexible, grâce à son architecture modulaire et à son support des extensions. Il peut s'adapter à de nombreux types d'applications et de besoins, notamment pour des applications géographiques avec des besoins plus complets.
- Par sa nature open source, il est compatible avec de nombreux langages de programmation, tels que Python, Java, C++, Ruby, PHP, etc.

Il est également important de noter que PostgreSQL tient sa popularité, au-delà de ses performances et fonctionnalités déjà complètes, grâce à sa capacité à gérer des types de données bien plus complexes. Il propose la gestion de modèles de données complexes tels que des données géographiques et des données attributaires, mais permet surtout de gérer les relations entre ces données. Cette gestion de données complexe permet une ouverture sur d'autres systèmes, particulièrement QGIS, un système d'informations géographiques, et ainsi d'étendre les fonctionnalités proposées par ce système. En type de fichiers volumineux, on peut par exemple citer les fichiers MAJICS, RPG, référentiels vecteurs, etc.

2.2.4. Les inconvénients de PostgreSQL

PostgreSQL présente également quelques inconvénients qu'il faut prendre en compte :

- Il peut être plus complexe à installer et à configurer que d'autres SGBDR, tels que MySQL ou SQLite.
- Il peut nécessiter plus de ressources matérielles (mémoire, CPU, espace disque) que d'autres SGBDR pour fonctionner de manière optimale.
- Il peut être moins performant que d'autres SGBDR pour certaines tâches spécifiques, telles que les requêtes de type OLAP (Online Analytical Processing).

« PostgreSQL 12 - Guide de l'administrateur » de Guillaume Lelarge et Stéphane Schildknecht, éditions Eyrolles, 2020.

« PostgreSQL - Maîtrisez les fondamentaux du SGBD open source » de Régis Montoya, éditions ENI, 2019.

« PostgreSQL - Le guide complet de l'administrateur et du développeur » de Joshua D. Drake et Peter Eisentraut, éditions Pearson, 2018.

2.2.5. Conclusion

PostgreSQL est un SGBDR open source très populaire, grâce à sa fiabilité, sa robustesse, sa richesse fonctionnelle et sa flexibilité. Il est utilisé dans de nombreux domaines, tels que la finance, la santé, l'éducation, le gouvernement, etc. Il est également compatible avec de nombreux langages de programmation et de nombreux systèmes d'exploitation. Cependant, il peut être plus complexe à installer et à configurer que d'autres SGBDR et nécessiter plus de ressources matérielles. Malgré ces inconvénients, PostgreSQL reste un choix idéal pour de nombreuses applications critiques et environnements complexes.

2.3. Problématique

L'application InfSuite s'appuie sur de nombreuses technologies, développées en externe de l'entreprise, pour fonctionner. Je peux par exemple citer Angular, un framework front-end développé par Google, qui est régulièrement mis à jour avec des réparations de bugs, des ajouts de nouveautés… et qui est utilisé dans l'application. Ce framework est continuellement mis à jour sans préavis pour les utilisateurs, mais utilise un système de versionnement. Le versionnement permet d'éviter les changements trop fréquents et perturbateurs, tout en offrant la possibilité d'introduire des améliorations et des nouvelles fonctionnalités de manière contrôlée. Il aide également les développeurs à anticiper les changements majeurs et à maintenir leurs applications à jour avec les dernières versions d'Angular.

Par exemple, dans le cas où Google déploierait des changements de code majeurs qui rendent le code ultérieur obsolète, le système de versionnement lui permet de rendre sa mise à jour accessible au grand public, sans pour autant forcer la main aux développeurs à effectuer une mise à jour qui pourrait mettre le fonctionnement de leur application en péril.

La majorité des technologies utilisées par InfSuite suivent ce système de versionnement. Malheureusement, réaliser de nouveaux développements pour l'application tout en surveillant les mises à jour des composants externes, analyser leur impact, les intégrer à l'application dès que ces dernières sont disponibles, n'est clairement pas possible. La stratégie de mises à jour utilisée est donc d'effectuer occasionnellement ces mises à jour, soit lors de moments clés (de nouvelles fonctionnalités importantes, des mises à jour de sécurité, des gains de performances, etc.), soit spontanément pour maintenir l'environnement à jour.

Ce cas de figure est présent à toutes les couches de l'architecture logicielle d'InfSuite, front-end, back-end, bases de données. Depuis un certain temps, l'application InfSuite utilise la version 14 du SGBDR PostgreSQL. Sortie en septembre 2021, cette version est maintenant dépréciée en faveur de la dernière version stable de l'application 16. Cependant, la version 14 a été adoptée très tôt par InfSuite et est restée depuis la version utilisée dans l'environnement InfSuite.

Pour suivre la stratégie de mise à jour de l'application, le sujet d'une mise à jour de la base de données d'InfSuite, de la version 14 vers la version stable la plus récente, a été mis sur la table. Le chef de projet en charge d'InfSuite a alors demandé une expertise pour la réalisation de cette migration pour s'assurer de la viabilité, mais également d'envisager, si possible, de réaliser cette migration en utilisant les nouveautés disponibles pour cette migration. Je vais donc chercher à savoir quelles sont les étapes préliminaires

pour s'assurer de la viabilité d'une telle migration, comment réaliser cette migration de manière optimale dans le contexte d'InfSuite et comment s'assurer de la qualité de la réalisation de cette dernière.

2.4. Existants

Effectuer une migration de base de données n'est pas une tâche anodine. Cela demande du travail en amont, il faut analyser les différents scénarios possibles, estimer un budget pour une telle tâche, réaliser de potentiels développements, s'assurer de la fiabilité avant de mettre tout cela en pratique et enfin la réalisation de l'étape cruciale sur les environnements sensibles. Pour faire une migration de base de données, il existe de nombreuses solutions, certaines plus coûteuses, d'autres plus fiables, encore d'autres plus spécialisées, etc. Il faut donc dans un premier temps trouver différents outils pour comparer les avantages et leurs faiblesses.

2.4.1. Les outils

On peut dans un premier temps penser à effectuer cette migration grâce à des outils spécialisés qui se chargent de faire la migration automatiquement et de s'assurer de la pérennité entre les données de l'ancienne base et les données insérées dans la nouvelle. En prenant pour exemple l'outil Oracle Data Dump fourni par l'entreprise Oracle, ce logiciel permet de sauvegarder et restaurer des données et des métadonnées pour les bases Oracle. Il est rapide et fiable, on peut ainsi lui fournir un fichier dump de la base source et l'outil va se charger, grâce à ce fichier, d'intégrer les données dans la base cible. Microsoft propose sa solution alternative SSMA pour importer les types de bases Access, DB2, MySQL, Oracle, SAP ASE vers leurs différents SGBDR propriétaires (à savoir les suites SQL Server). Sur le même principe, les outils MySQL Workbench, AWS Database Migration Service (DMS) et PgAdmin permettent de réaliser le même type de migration vers leurs systèmes propriétaires, à savoir respectivement MySQL, AWS et PostgreSQL.

Les principaux désavantages de ce genre de solutions sont :

- L'import des données reste assez strict et ne permet pas de flexibilité. Si les données ne sont pas compatibles, il faut passer du temps à travailler le modèle de données pour essayer de pallier aux incompatibilités.
- Ils peuvent également rendre les migrations onéreuses avec certaines solutions qui coûtent jusqu'à plusieurs centaines d'euros pour les entreprises.
- Les logiciels proposés peuvent parfois être complexes et demander un certain temps d'adaptation avant de réellement pouvoir effectuer la tâche de migration.

2.4.2. Les types de migrations

Il est possible, comme vu précédemment, d'effectuer une migration **à partir d'outils automatiques**. L'avantage est de déléguer la majorité de la complexité à une application qui va se charger d'effectuer la tâche cruciale et de réduire les risques d'erreurs pour de grosses applications. On retrouve en général des outils plus poussés pour offrir une

plus grande précision et une meilleure cohérence dans la migration des données. Il faut principalement noter, pour ce genre d'outils, qu'ils sont en général destinés à un certain type de base précis et qu'ils peuvent donc manquer de compatibilité. Ils peuvent également ne pas prendre en charge tous les types de bases de données ou tous les scénarios de migration, ce qui peut limiter leur utilité dans certains cas. Enfin, le coût réel de ces outils peut être élevé, autant par leur prix réel fixé par l'éditeur, mais aussi parce que la majorité du temps, ils nécessitent une expertise pour être pris en main avant de pouvoir être réellement utilisés.

Une alternative à prendre en compte est la migration **manuelle**. Cette solution est intéressante pour les petites bases de données sans trop de complexité. Le but est d'exporter un fichier dump de la base source et de l'importer dans la base cible si possible, ou d'exporter les données sous un autre format pour l'importer simplement. Les coûts de cette technique sont faibles, voire nuls et permettent une flexibilité lors de la migration. Cependant, le temps de migration peut se révéler très long, apporte un gros facteur de risque d'erreurs et devient complexe, voire inapproprié dans le cadre de bases avec de gros volumes.

Une autre technique consiste à effectuer la migration **via des scripts**. Le but est de développer un processus qui va récupérer les données de la base source, effectuer si besoin des opérations sur les différentes données pour les copier dans la base cible par la suite. Quasiment tous les langages de programmation permettent de créer des scripts pour effectuer ce genre de migration, il suffit qu'un driver pour les bases utilisées soit disponible pour le langage souhaité. L'avantage de ce type de migration réside dans la flexibilité totale pour personnaliser le processus de migration. Il est également possible de migrer des bases de données plus grandes et plus complexes avec des incompatibilités entre elles, puisqu'il est possible d'agir sur les données avant de les transférer vers la nouvelle base, ce qui confère un contrôle total sur la migration. Cependant, il faut noter que les coûts de la migration peuvent aussi devenir élevés parce qu'il faut développer un script, donc payer un développeur. Il faut aussi prendre en compte que le temps de migration peut être long puisqu'il faut développer la solution en amont et qu'il faut en général s'assurer de la qualité du code avant de l'utiliser sur les environnements sensibles. Même en essayant de prévenir les différents cas d'erreurs possibles, il se peut que certaines arrivent à se glisser, dans ce cas, le risque d'erreur humaine n'est pas négligeable.

Enfin, une solution peut se porter sur **la migration en temps réel**. La migration en temps réel est une technique qui permet de répliquer les données de la base source vers la base cible en temps réel, sans interrompre les opérations sur la base source. Elle est souvent utilisée pour minimiser le temps d'arrêt lors de la migration de bases de

données critiques (secteur de la santé, du commerce, de la sécurité, etc.). Ce processus de migration en temps réel implique les mêmes coûts qu'une migration par script ou par outils automatiques puisqu'elle va se reposer sur l'un de ces deux piliers. Elle va cependant permettre de minimiser l'impact sur le service actif parce qu'elle ne requiert généralement pas de mettre le service à l'arrêt. Il faut cependant noter que lors de la migration, l'impact des erreurs pouvant se glisser durant le processus de migration deviendrait rapidement beaucoup plus important, car les données sont consommées à l'instant où elles sont migrées. La migration en temps réel n'est donc réellement intéressante que dans le cas où la relation entre une application qui ne peut pas avoir de temps d'inactivité avec les données de la base est critique.

2.5. Conclusion

Il existe de nombreux outils pour effectuer des migrations de données à partir de bases de données. Cependant, il faut prendre en compte différents critères, comme la complexité de la migration, le budget alloué, les systèmes source et cibles, mais aussi la stratégie à appliquer lors de cette étape. Par exemple, dans un cas, le temps alloué à la migration peut être ignoré si ce qui compte est d'effectuer la migration sans arrêter le service, dans un autre cas, le temps d'arrêt est moins important que la fiabilité de la migration. Il faut donc réfléchir en amont à la manière d'effectuer cette étape, mais surtout à la pertinence de cette tâche pour éviter des coûts supplémentaires qui pourraient se révéler inutiles. Toutes ces questions permettent de cibler le besoin et donc le type de migration souhaité pour, par la suite, transférer ses données entre deux systèmes. Je peux maintenant m'intéresser à la place de ce SGBDR au sein de l'application InfSuite.

3. Réalisation

3.1. Introduction

Je présente dans cette section le travail réalisé durant les neuf mois au sein de l'entreprise Unit Solutions. Portant sur le sujet d'une migration de base de données dans le but de maintenir les technologies de l'application à jour, je présente tout d'abord les différents éléments qui justifient le choix d'effectuer cette migration de base de données, par la suite, j'expose l'analyse préliminaire réalisée pour s'assurer que la migration reste pertinente, j'aborde plus tard les changements réalisés dans le code et enfin, je présente les résultats obtenus post-migration.

3.2. Pertinence et philosophie

Lorsque le sujet de mettre à jour le SGBDR a été présenté, il n'a pas été décidé de remplacer PostgreSQL. Comme expliqué précédemment, PostgreSQL excelle dans la gestion de données géographiques et attributaires, il est complet et surtout très bien maintenu par ses développeurs. Ces différentes raisons en font un candidat parfait pour l'environnement InfSuite plus que tout autre système proposé par la concurrence.

Maintenir PostgreSQL en tant que SGBDR de l'application écarte déjà un bon nombre de solutions au niveau logiciel. Je peux d'ores et déjà écarter les logiciels payants et longs à maîtriser autres que PgAdmin pour effectuer cette migration, qui reste le logiciel le plus adéquat puisque nous ne cherchons pas à changer d'environnement.

L'application InfSuite souhaite utiliser une fonctionnalité importante de PostgreSQL : les timestamps, ou horodatages en français.

En effet, InfSuite utilise un système de version basé sur de l'horodatage. Quand un utilisateur met à jour les informations d'un document dans l'application (inspection, restauration, etc.), il crée une nouvelle version du document.

Si deux utilisateurs récupèrent la même version d'un document, qu'ils effectuent tous deux des mises à jour sur ce document, que l'utilisateur A sauvegarde ses changements en premier, que l'utilisateur B essaye de sauvegarder a son tour le document avec ses changements, alors le serveur lui signalera qu'il ne possède pas la dernière version du document, qu'il a été modifiée entre-temps et donc qu'il ne peut pas sauvegarder ses modifications sans mettre à jour la version de son document.

Pour assurer ce suivi de version, le document contient une version en base, qui est un timestamp sans fuseau horaire. Un problème peut alors apparaître si deux utilisateurs, sur des fuseaux horaires différents, effectuent des modifications au même instant.

Le serveur ne pourra pas vérifier l'information de fuseau horaire puisqu'elle n'est pas sauvegardée en base et donc, si l'utilisateur effectue une sauvegarde même avec une ancienne version du document, cette dernière écrasera les derniers changements.

Dans le schéma Fig. 5, deux utilisateurs génèrent des versions de fichiers différentes. Un utilisateur enregistre ses changements en premier et génère la version AB du document. Quand le second utilisateur souhaite enregistrer ses changements C, un conflit de fichier l'en empêche. C a comme origine la version A et non pas AB, l'utilisateur doit donc récupérer les changements de la version AB avant d'enregistrer ses changements.

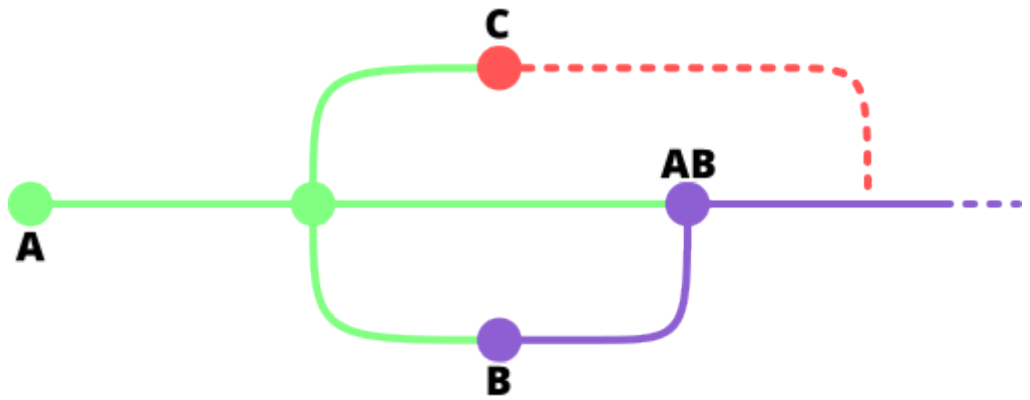

Fig. 5. – Schéma d'un conflit de version entre deux version d'un document

De plus, un point important pour lequel le souhait de mettre à jour la base de données a été exprimé est de pouvoir profiter des améliorations du SGBDR PostgreSQL. Il y a par exemple des mises à jour de sécurité du système. Mettre à jour une application et ses différents composants permet de profiter des mises à jour de sécurité afin d'éviter des vulnérabilités qui pourraient mettre en péril les données des clients et l'intégrité de l'entreprise.

Les vulnérabilités sont connues sous le nom de Common Vulnerabilities and Exposures et, en prenant pour exemple la mise à jour pour corriger la faille CVE-2023-5869, PostgreSQL résout un problème permettant d'écrire des octets arbitraires en mémoire, qui facilitait l'exécution de code arbitraire. Normalement, les ORM permettent de faire automatiquement de la paramétrisation de requêtes pour prévenir d'injections potentielles. Mais il se peut que des failles supplémentaires se cachent dans le code et permettent donc, malgré tout, d'exploiter ces failles. Les corriger sur le système permet alors d'éliminer en grande partie le risque d'attaque.

En complément des vulnérabilités, les mises à jour apportent des améliorations de performances cruciales. En théorie, les ORM jouent un rôle essentiel dans les performances d'une application, car ils utilisent un cache pour éviter d'effectuer des requêtes vers la base de données à trop haute fréquence.

Dans le cas de la base PostgreSQL, elle est aussi utilisée pour le système d'information GIS, il y a donc beaucoup de données à délivrer et avec un grand nombre de requêtes. Améliorer les performances permet de réduire la latence ressentie par l'utilisateur lorsqu'il cherche à charger des données dans l'application. Sur la version 16 de PostgreSQL, il est promis une amélioration de 10 % des performances du SGBDR. Un tel gain de performance permet d'améliorer grandement l'expérience utilisateur.

Il faut maintenant déterminer quel est le type de migration le plus adéquat dans ce cas de figure pour éviter les coûts supplémentaires qui peuvent être facilement évités.

Je dois dans un premier temps chercher les technologies utilisées dans l'application et m'assurer que celles-ci soient compatibles avec la nouvelle base. Par la suite, je vais devoir valider la compatibilité entre la version de la base source et de la base cible. Avec cette confirmation, je pourrai alors chercher la meilleure manière d'effectuer cette migration parmi les différentes options disponibles que j'ai pu exposer précédemment.

3.2.1. Analyse préliminaire

En regardant côté architecture, je cherche quelles sont les applications qui risquent de rencontrer un problème lors de la migration. Les seuls systèmes qui établissent une relation directe avec la base de données sont côté back-end. On limite donc l'impact sur l'application globale s'il y a des changements et des mises à jour à réaliser pour permettre à l'application de continuer à fonctionner avec la base de données.

Le projet back-end de l'application est développé en C# avec .NET qui gère la partie web de l'application. Un certain nombre de librairies sont utilisées par le projet InfSuite pour permettre d'étendre les fonctionnalités de l'application sans devoir dépenser de temps de développement.

Il existe des paquets alimentés par Microsoft, par exemple comme .NET, Entity Framework, … et d'autres librairies qui sont elles développées par des développeurs tiers qui ont souhaité ajouter une nouvelle fonctionnalité à l'écosystème.

Quand on souhaite effectuer une mise à jour de l'application, l'environnement Microsoft nous propose d'effectuer la mise à jour de ces librairies nommées NuGet Packages dans l'écosystème C# via NuGet, le gestionnaire de dépôts associé.

Il permet de faire du versionnement des librairies, sur le même principe que PostgreSQL qui utilise le concept de versionnement pour livrer ses mises à jour.

Chaque version d'une librairie peut notifier des incompatibilités avec certaines versions d'autres librairies. Je dois donc étudier les composants qui permettent de faire la liaison à la base de données et également les librairies dont peut ou qui peuvent dépendre de ces composants.

Il faut prêter attention à chaque composant mis à jour indépendamment qui pourrait causer d'autres problèmes de compatibilité avec l'application et casser des fonctionnalités.

Un projet C# dans l'environnement Microsoft est composé d'une solution, de sous-solutions et de projets.

Les sous-solutions d'un projet sont des solutions indépendantes intégrées au projet. Un peu à la manière de librairies, je peux créer du code qui pourra être réutilisé dans différents projets indépendamment, sans devoir le dupliquer.

Chaque projet possède ses propres NuGet packages, mais la solution peut gérer des dépendances, c'est-à-dire que le package sera ajouté à la liste des dépendances globales et sera installé dans tous les projets de la solution.

Dans mon cas, la solution back-end d'InfSuite est composée de 65 projets. Il y a le projet « WebApi » qui gère les routes et les interactions avec l'application front-end qui est disponible pour le client et qui dépend du projet « Server ». Le projet « Server », lui, contient la logique de certaines interactions simples, par exemple la recherche d'IO via la barre de recherche dans l'application. Le projet « Server » dépend du projet « Core » qui gère certaines fonctionnalités plus poussées de l'application autant pour la partie web que pour d'autres composants. Il existe de nombreuses autres dépendances dans l'application qui complexifient la gestion des packages. Par exemple, dans le diagramme Fig. 6, on remarque rapidement que les dépendances internes au projet peuvent être relativement complexes.

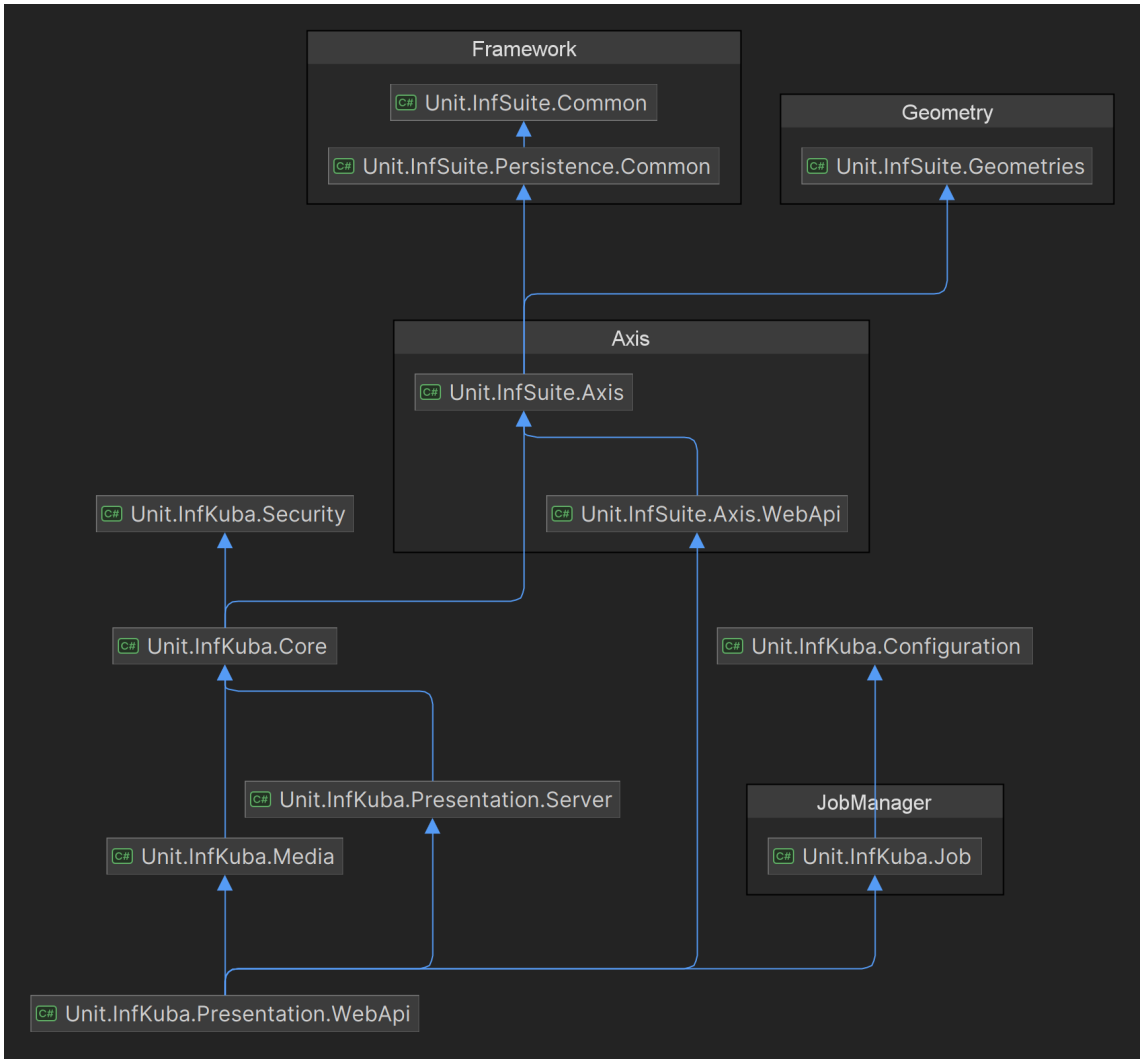

Fig. 6. – Arbre de dépendances du projet WebApi d'InfSuite

Si je souhaite par exemple mettre à jour un package dans le projet « Geometry », tous les projets dépendants sont impactés par cette mise à jour, il faut donc s'assurer que rien ne vient casser le bon fonctionnement de l'application en profondeur. Je m'intéresse donc aux liens entre les différents projets ainsi qu'aux librairies que je souhaite mettre à jour pour m'assurer de leur compatibilité pour toute l'application.

3.2.2. Compatibilité et décision

La première étape est de valider la compatibilité entre la version 14 et 16 de PostgreSQL. Une première manière de valider cela est d'aller sur les notes de mise à jour sur le site officiel et de chercher les changements qui peuvent affecter la compatibilité entre les deux bases.

Après avoir analysé en détail les changements, il y a trois grandes catégories. Il y a l'impact sur l'administration système, celui sur les requêtes SQL et celui sur l'incompatibilité des données. La partie administration système n'est pas gérée par les

développeurs d'InfSuite, elle est gérée par d'autres employés de l'entreprise, je peux donc me concentrer sur la partie requêtes et données.

Pour la partie des données stockées, en analysant les données actuelles, je peux relever quelques changements qu'il faut surveiller :

- Les jsonb peuvent être utilisés par les systèmes d'informations géographiques pour stocker les informations géographiques et attributaires. Il faut donc surveiller le comportement de l'application à ce niveau.
- Les uuid qui ont subi des optimisations pour mieux gérer les identifiants uniques universels et qui sont utilisés comme identifiants uniques pour les tables dans le projet.
- Les types numeric qui sont également utilisés dans l'application et qui ont subi des optimisations de performances.

Il n'existe pas d'autre changement majeur qui pourrait porter atteinte à l'intégrité des données lors de la migration.

Une fois la première étape validée pour l'intégrité théorique des données, il faut valider la seconde étape qui est la réalisation de la migration. Le seul moyen de s'assurer de la complète compatibilité entre les deux systèmes est d'effectuer une première migration dite « à blanc ». Elle a pour but de mettre en place un système séparé de l'environnement de production, pour s'assurer de ne pas impacter l'application accessible par les clients.

L'objectif de ce système est d'avoir une base de tests sur laquelle je peux m'assurer que l'application fonctionne toujours après la migration de données et que si des problèmes s'annoncent, il n'y ait aucun autre impact que de les constater.

Avec la validation obtenue précédemment sur la compatibilité des deux systèmes, j'ai pu valider, toujours avec l'approbation de mon chef de projet et de l'architecte d'application, que la migration la plus adaptée dans notre cas serait une migration hybride.

Le but de la migration hybride est de combiner deux types de migration de données que j'ai pu citer précédemment dans ce document. Comme je veux ajouter un fuseau horaire aux horodatages qui servent de versions, je dois modifier les données de la base source. Cependant, cela implique de modifier les données et les modèles de données associés.

Pour éviter de devoir utiliser un logiciel tel que PgAdmin et de passer du temps à manipuler le modèle de données qui pourrait causer des problèmes, je préfère utiliser des outils bien maîtrisés et intégrés sur le projet InfSuite.

Pour l'import de données, cette tâche a été réalisée par un administrateur système de l'entreprise. Il a utilisé la méthode d'import d'un fichier dump de la base source dans la nouvelle base. Je l'ai présenté ci-dessus, n'ayant aucun souci de compatibilité sur les types de données, l'intégration de ces données s'est faite sans accroc dans la base de test.

Pour mettre à jour le modèle de données de manière synchrone avec les mises à jour de l'application et éviter des soucis de compatibilités, l'environnement InfSuite utilise un système de fichiers scripts. Rédigés en SQL, ces scripts viennent mettre à jour le schéma de base de données et sont exécutés automatiquement par le serveur lors de la mise à jour de l'environnement ciblé (dev, staging, production, etc.).

Pour l'exemple des horodatages, je rédige un script SQL que je pourrais exécuter manuellement sur l'environnement de développement avec la nouvelle version de PostgreSQL et effectuer mes tests. Cependant, si je veux intégrer mes modifications sur les autres environnements, il faut que je sauvegarde mes changements sur Azure DevOps, je n'ai pas la permission de faire ces changements manuellement.

Enfin, pour mettre à jour les données dans la base une fois le schéma de base modifié, je décide de faire cette migration en temps réel. La compatibilité entre les types de données timestamp et timestamptz permet de garder en base l'ancien format de version, de mettre à jour le schéma de base puis de mettre à jour les données en y rajoutant le fuseau horaire. Comme la colonne « version » est utilisée à de nombreux endroits, créer un type de donnée côté serveur qui va s'occuper de les traiter rend la tâche moins complexe et chronophage. Avant de passer à la réalisation, je dois m'assurer de la compatibilité des dépendances externes pour effectuer la migration.

Ce qui m'intéresse particulièrement ici, est de mettre à jour le package Npgsql car il permet de communiquer avec la base de données. Dans une application .NET, il y a la possibilité d'utiliser un Object Relational Mapping pour simplifier les interactions avec les bases de données. Le projet InfSuite utilise historiquement l'ORM EntityFramework. Il faut bien différencier EntityFramework qui n'est plus activement développé, de son successeur EntityFrameworkCore qui offre de nouvelles fonctionnalités qui ne seront plus implémentées dans EntityFramework.

EntityFramework utilise le connecteur Npgsql pour effectuer les requêtes vers la base de données. En recherchant les versions disponibles de Npgsql pour EntityFramework, je me suis rendu compte qu'il fallait prêter attention à une fonctionnalité importante de PostgreSQL que le projet InfSuite veut maintenant exploiter : les horodatages avec des fuseaux horaires.

La version post mise à jour de Npgsql était la version 4.1.13 et la dernière version disponible qui utilise les horodatages avec fuseau horaire est la version 8.0.3. En essayant de le mettre à jour avec cette version, je me suis rendu compte qu'EF n'était pas compatible avec la version 8.0.3. La dernière version du paquet Npgsql disponible pour EF étant la version 6.4.3, je dois donc me contenter de cette version.

Avant de valider l'utilisation de ce dernier dans toute l'application, je dois valider qu'elle supporte les horodatages avec fuseau horaire et qu'elle est également compatible avec le reste de l'application. En se référant à la documentation fournie, la première version à intégrer les horodatages avec fuseau horaire est bien la version 6.X.X d'Npgsql, elle est donc compatible avec le besoin de sauvegarder les fuseaux horaires pour les versions.

Si on souhaite passer sur la nouvelle version du paquet Npgsql, la version 8.X.X, il n'y a pas d'autre choix que de passer d'EF à EFCore. Faire le choix de mettre à jour l'application EFCore permettrait de compléter l'objectif de maintenir l'application à jour et saine. En effet, pour rappel, EF n'est maintenant plus mis à jour régulièrement.

L'étude d'impact des coûts et des changements impliqués par une telle mise à jour a déjà été réalisée à priori de mon analyse et il en est ressorti que pour l'année courante, un tel budget ne pouvait pas être accordé pour cette mise à jour. Si je veux mettre à jour l'application pour utiliser les dernières fonctionnalités disponibles sans passer par une mise à jour d'EF, le seul choix qu'il reste est de mettre à jour Npgsql vers la version 6.X.X.

Le schéma Fig. 7 ci-dessous résume la situation d'incompatibilités entre les versions des trois éléments à savoir respectivement : à gauche Entity Framework, au centre le paquet Npgsql et à droite la base de données PostgreSQL.

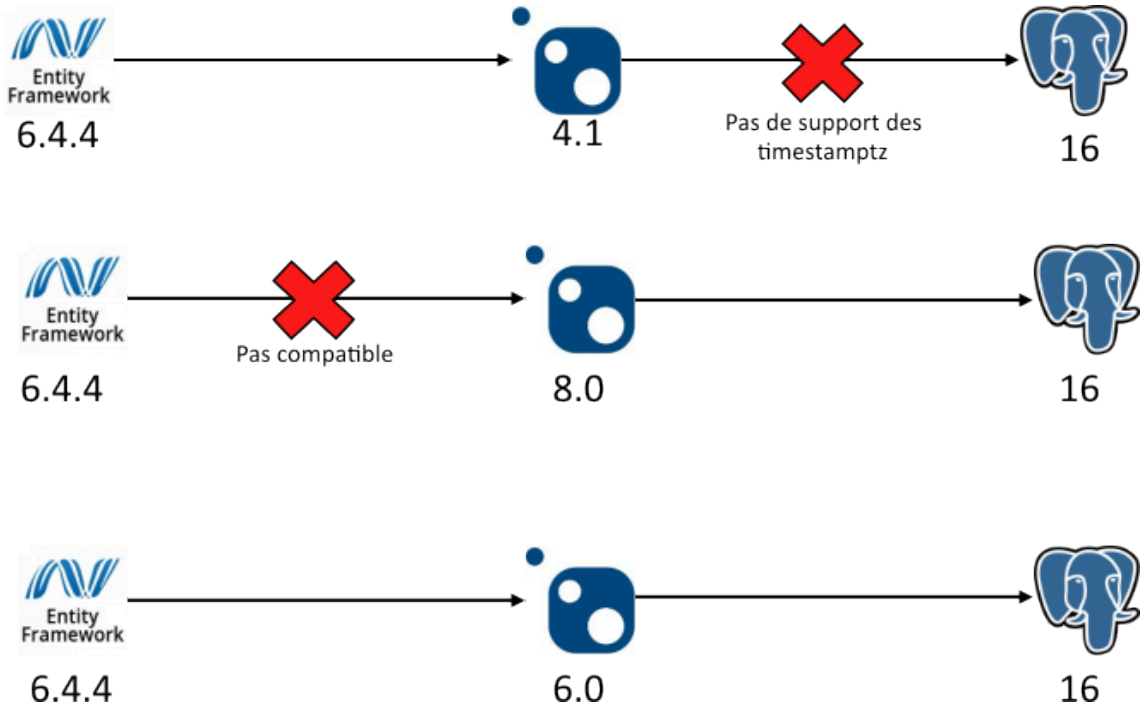

Fig. 7. – Problématique d'incompatibilité entre les différents éléments permettant au serveur et à la BDD d'échanger des informations.

3.2.3. Tests de performance

La dernière étape permettant de valider cette étape de migration est de s'assurer que la nouvelle version de PostgreSQL installée à de meilleures performances que la version remplacée. Pour valider cela, il a été développé un système en interne de test de performance. L'architecture complexe de la base et la relation avec des données géographiques et attributaires peut rendre les requêtes parfois plus longues d'exécution.

Pour effectuer un test de performance, le système utilise une fonctionnalité gourmande en ressource, les groupes. Dans l'environnement InfSuite, il est possible d'afficher sur la carte les objets d'infrastructure. Par défaut tous les objets d'un dataowner sont affichés. Pour permettre d'axer son travail sur des objets d'infrastructures particuliers, alors il est possible d'appliquer un filtre à toutes ces entités. Le mode « Groupe » de l'application permet de créer alors deux types différents de filtres.

Le premier type de filtres est un filtre statique. Peu gourmand en ressource et très simple d'utilisation, on peut y ajouter des entités manuellement et cela va permettre de n'afficher que les objets sélectionnés dans l'application. Fonctionnellement parlant ce n'est qu'une relation de base de données.

Le second type de filtre, plus complexe, est le filtre dynamique. Les filtres dynamiques permettent de gérer automatiquement les entités qui seront incluses ou exclues de l'affichage. Pour ce type de filtre, il est possible de créer des règles de filtrage avancées pour permettre à l'utilisateur plus de flexibilité.
Un peu à la manière d'une requête SQL, il est possible de construire une requête qui va être exécutée pour permettre de retrouver les entités à partir de critères. Il est par exemple possible de demander de filtrer les ouvrages par position géographique en ne prenant en compte que ceux au-dessus d'une certaine latitude et de rajouter à ce filtre uniquement les ouvrages qui contiennent un certain numéro dans leur nom.
Il est possible de créer des règles avec chaque propriété d'un objet d'infrastructure, d'appliquer un ordre pour les conditions, de faire des agrégats, des jointures,… Les utilisateurs ont la liberté de créer leur propre requête, une requête pouvant devenir rapidement complexe, elle peut donc prendre plus de temps à filtrer les objets d'infrastructures pour les retourner à l'utilisateur.
Étant utilisé par les services GIS de l'application, s'assurer de la performance de ce système est primordial.

Ce que l'outil permet de tester est ce système de groupe/filtres. Il va chercher les différents filtres existants en base de données, peu importe le client qui a pu les créer, et les exécuter.

Cette méthode permet de connaître les performances réelles de la base dans un cas pratique spécifique à l'application et non pas d'avoir des performances théoriques fournies par les développeurs du SGBDR qui peuvent être tournées en leur faveur. Nous cherchons donc à savoir si le chiffre de 10% en gain de performances est réel ou non.

L'objectif est donc de comparer les performances de la base de production encore sous PostgreSQL 14, avec les performances d'une base de développement installée pour l'occasion, elle, sous PostgreSQL 16. Après avoir exécuté l'outil de benchmark sur les deux bases, je récupère les données brutes en sortie de programme pour les analyser. Ci-dessous un exemple de données de sorties fournies par le programme après exécution.

```
eruid,description
batman,uses technology
superman,flies through the air
spiderman,uses a web
ghostrider, rides a motorcycle
#GROUP_OBJECT_PROFILE#accessgroupGroupProfile
cn,description
daredevil,this group represents daredevils
superhero,this group represents superheroes
#GROUP_OBJECT_PROFILE#aixaccessgroupGroupProfile
aixgroupadminlist,ibm-aixprojectnamelist
eadmins,eadmingroup
eguests,eguestgroup
```

Les données sont au format CSV, je peux ainsi les importer dans un logiciel tableur tel qu'Excel pour les manipuler et les comparer. L'application de production étant accessibles aux clients de manière continue, les données stockées peuvent changer très rapidement et présenter un delta avec les données sauvegardées à posteriori lors la mise en place de la base de développement.
Cette différence doit être prise en compte dans la comparaison des données puisque je dois alors utiliser des fonctions plus poussées d'Excel pour retrouver les données similaires entre les deux tables et en comparer les résultats.

J'obtiens dans les données les informations d'identification des groupes testés, le client à qui ils appartiennent, le nombre d'ouvrages que les groupes contiennent après filtrage, le temps total d'exécution du groupe et le temps moyen calculée pour le chargement d'un objet d'infrastructure. L'unité de temps est donnée en millisecondes.

Pour comparer les performances de la base, je commence par récupérer les données globales des deux bases, à savoir le nombre total d'objets d'infrastructures qui ont étés traités et le temps global d'exécution de traitement. Pour connaître le temps total qu'a mis la base pour traiter tous les groupes, j'utilise la requête

`=SUM(pg14[Load_Duration_In_Ms])` qui fait une simple somme de tous les résultats de la colonne `Load_Duration_In_Ms`. Je fais la même chose pour les résultats des deux bases et pour la colonne `Count_Ios` que je réutilise avec un simple calcul de différence dans la requête `=Total_Ios_Count_14 - Total_Ios_Count_16`. Comme le présente le tableau des résultats Tableau 1, les données dans la base de production ont déjà changées, il faut donc que je tienne compte dans mes résultats.

	Postgres_14	Postgres_16	Gap_From_14_To_16
Total_Io_Count	323527	318634	−4893
Total_Duration	3577961	3450208	−127753
Identical_Ios_Duration	3499476	3438261	−61215
Average_Ios_Load_Duration	11,05923462	10,82812255	−0,231112075

Structure du tableau de résultats:

- La colonne **`Gap_From_14_To_16`** contient les résultats des comparaisons des données obtenues entre Postgres 14 et Postgres 16.
- La ligne **`Total_Io_Count`** indique le nombre total d'objets d'infrastructures qui ont été filtrés par les différents groupes traités.
- La ligne **`Total_Duration`** indique le temps total de traitement (en millisecondes) de tous les groupes.
- La ligne **`Identical_Ios_Duration`** indique le temps de traitement total (en millisecondes) uniquement pour les OIs identiques entre les deux bases.
- La ligne **`Average_Ios_Load_Duration`** indique le temps de traitement moyen écoulé (en millisecondes) par objet d'infrastructure.

Dans les réultats obtenus, je remarque rapidement que le temps de traitement entre les deux bases n'est pas drastiquement différent. Pour le temps de traitement global des objets d'infrastructure identiques entre les deux bases, on ne gagne que 0.23 milliseconde après la migration. Pour filtrer mes résultats j'utilise la méthode `=SUM(SUMIF(pg14[Group_Id]; VLOOKUP(pg14[Group_Id]; pg16[Group_Id]; 1;FALSE); pg14[Load_Duration_In_Ms]))` qui va exclusivement récupérer les lignes où les groupes anaylsés sont présents dans les deux bases puis faire la somme des résultats de performances.

Pour terminer mon analyse, je transforme les résultats en pourcentages dans le tableau Tableau 2. Avec ça je peux comparer les 10% de performances annoncées avec les résultats réels obtenus.

Identical_Group_Count	3028
Identical_Ios_Perf_Gain	1.78%
Total_Perf_Gain	3.70%

Comme je l'ai remarqué rapidement, le gain réel de performances entre les deux bases n'est que de 3.7%. Le score affiché pour les données anaylsées identiques entre les deux bases est encore plus faible avec uniquement 1.78% de perfomances gagnées.

Ce résultat peut être le résultats de nombreux facteurs. Une première hypothèse qu'on peut emettre est que l'analyse de performances donnée par PostgreSQL concerne d'autre types de requêtes ou de perfomances de base. Un autre cas de figure est que le temps de traitement le plus long dans notre cas est obtenu par l'interface entre le code et la base: EntityFramework. En effet il se pourrait que la gestion des requêtes côté ORM soit la plus couteuse en ressource que tout le reste du système ce qui impacte le résultat final. Une dernière hypothèse serait un biais dans mon analyse. Une erreur humaine est plausible et il n'est pas exclu que je me sois trompé dans mes requêtes ou que j'ai oublié de prendre en compte tout autre élément ayant un impact direct sur les performances obtenues.

J'ai donc pu rendre mon rapport sur les comparatifs de perfomances à mon chef de projet pour qu'il valide, ou non, la suite de la migration de base de données. Les résultats de perfomances obtenus seuls, ne permettent pas de justifier un tel investissement pour cette migration. Cependant, comme évoqué plus haut, d'autres points importants tel que la sécurité ou la pérennité de l'application en mettant à jour la base on permit de soutenir la décision de continuer la migration. J'ai donc pu continuer les étapes de migration en passant sur la partie d'adaptation du code pour permettre à l'application de fonctionner avec la nouvelle base.

3.3. Adaptation du code

Dans l'application, 24 projets au total utilisent le paquet Npgsql. Ces projets ne sont pas uniquement dédiés à la partie back-end de l'application, mais sont également des outils tiers développés pour des besoins spécifiques. Je peux par exemple citer l'outil IkCoordToRvg que j'ai pu aborder dans mon mémoire de licence, qui était le projet sur lequel j'ai pu travailler durant toute une année. Comme pour tout autre projet, il utilise des dépendances au projet Core d'InfSuite, qui nécessite lui-même le paquet Npgsql. Lorsque j'ai tenté une première fois de mettre à jour le paquet, je me suis retrouvé confronté à de nombreuses erreurs de compatibilité avec des dépendances externes.

Pour régler les conflits, une simple mise à jour vers une version plus récente des paquets concernés a suffi. Tout comme pour la mise à jour Npgsql, je m'assure dans un premier temps qu'il n'y ai pas de changements majeurs qui risque de casser le bon fonctionnement de l'application, puis j'effectue la mise à jour. Je dois maintenant adapter le code concerné.

Pour gérer les formats de certaines données, l'application utilise une inférence du type de données. Pour le cas de la mise à jour de la colonne version, on la met à jour pour utiliser des horodatages avec fuseau horaire. Cependant, le reste des modèles utilisant des horodatages dans d'autres parties de l'application n'ont pas été mis à jour. Ils utilisent toujoursla versions sans fuseau horaire.

Il faut donc préciser à notre application que pour le type de données DateTime dans l'application InfSuite, cela correspond au type DateTime2 en base de donnée (horodatage sans fuseau horaire). Dans l'autre sens, il n'y a pas la même problématique puisque les DateTime en C# contiennent un fuseau horaire.
Ainsi, si au moment de la conversion l'horodatage ne contient pas cette information, alors le serveur utilise par défaut le fuseau horaire de sa propre localisation.

Pour gérer l'inférence des données, je dois rajouter une nouvelle étape de vérification pour le convertisseur JSON en lui précisant comment gérer le nouveau format. Un simple ajout de la ligne ci-dessous dans les types personnalisé permet cette inférence. Elle récupère simplement le contenu textuel de la requête et essayer de valider que c'est une date. Si c'est le cas alors, il infère le type Date pour l'application, sinon il exécute le reste du programme comme normalement.

```
JsonTokenType.String when reader.TryGetDateTime(out DateTime datetime) =>
DateTime.SpecifyKind(datetime, DateTimeKind.Unspecified),
```

Pour préciser quels sont les entitées en base de données qui n'utilisent pas le format par défaut DateTime2, je rajoute à la variable `Version` dans les entitées concernées

l'anotation `[JsonConverter(typeof(DateTimeUtcConverter))]`. De la sorte ,je le force à utiliser un convertisseur différent.

Pour permettre de gérer ces dates différement j'ai pu créer deux convertisseurs de dates. Ils ont un fonctionnement similaire, il ne varient que par le type de date retourné. Dans le bloc de code Fig. 8 les dates sont retournée sans fuseau horaire s'il n'est pas présent tandis que dans le bloc de code Fig. 9 il prends en valeur par défaut le fuseau horaire UTC.
Il est bon de remarquer que j'utilise dans mon anotation de variable pour mon entité ci-avant, le convertisseur que j'expose dans le bloc de code Fig. 9.

```
public class DateTimeUnspecifiedConverter: JsonConverter<DateTime>
{
    public override DateTime Read(ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions options)
    {
        if (!reader.TryGetDateTimeOffset(out DateTimeOffset datetime))
            return default;
        return DateTime.SpecifyKind(datetime.DateTime, DateTimeKind.Unspecified);
    }
    public override void Write(Utf8JsonWriter writer, DateTime value, JsonSerializerOptions options) ⇒
writer.WriteStringValue(value.ToString("yyyy-MM-ddTHH:mm:ss"));
}
```

Fig. 8. – Bloc de code d'un convertisseur de date sans fuseau horaire

```
public class DateTimeUtcConverter : JsonConverter<DateTime>
{
    public override DateTime Read(ref Utf8JsonReader reader, Type typeToConvert, JsonSerializerOptions options)
    {
        if (!reader.TryGetDateTimeOffset(out DateTimeOffset datetime))
            return default;
        return datetime.UtcDateTime;
    }
    public override void Write(Utf8JsonWriter writer, DateTime value, JsonSerializerOptions options) ⇒
writer.WriteStringValue(value.ToUniversalTime().ToString("yyyy-MM-ddTHH:mm:ssZ"));
}
```

Fig. 9. – Bloc de code d'un convertisseur de date avec fuseau horaire

Pour m'assurer que mes changements sont corrects et que je n'ajoute pas de code qui pourrait mettre en défaut l'application, je créais pour toute l'application des tests qui permettent de vérifier que la conversion fonctionne comme attendu. J'ai rajouté des tests manquants, avec le test d'inférence des types comme dans le bloc de code Fig. 10, pour vérifier qu'en fonction du continu le convertisseur JSON retrouve le bon type de données. J'ai également ajouté des tests pour les deux types de convertisseurs de données en lui fournissant différentes formes de dates et en m'assurant que ce qu'il me retourne, correspond à une date bien formée avec ou sans fuseau horaire en fonction du contexte.

```
public class InferredTypeConverterTest
{
    [Theory]
    [InlineData("true", true)]
    [InlineData("false", false)]
    [InlineData("123", 123L)]
    [InlineData("3.14", 3.14)]
    [InlineData("\"2024-02-28T15:30:45\"", "2024-02-28T15:30:45", typeof(DateTime))]
    [InlineData("\"6F9619FF-8B86-D011-B42D-00C04FC964FF\"", "6f9619ff-8b86-d011-b42d-00c04fc964ff", typeof(Guid))]
    [InlineData("\"sample string\"", "sample string")]
    [InlineData("null", null)]
    public void Read_JsonValue_CorrectlyConvertsToExpectedType(string jsonValue, object expectedValue, Type
expectedType = null)
    {
        var reader = new Utf8JsonReader(Encoding.UTF8.GetBytes(jsonValue));
        var converter = new InferredTypeConverter();
        while (reader.Read())
        {
            var result = converter.Read(ref reader, typeof(object), new JsonSerializerOptions());
            Assert.Equal(
                (expectedType ≠ null ?
                    TypeDescriptor.GetConverter(expectedType).ConvertFromInvariantString(expectedValue.ToString()) :
                    expectedValue),
                result);
        }
    }
}
```

Fig. 10. – Bloc de code d'un test d'inférence JSON

Mes derniers ajouts portent sur des modifications mineures comme des changements dans les logs de l'application pour faciliter le débogage, des ajouts de type de date dans les différentes énumérations concernées, etc.

3.4. Résultats

3.4.1. Performances finales

Durant le processus de modification du code, je me suis rendu compte que j'avais omis un détail lors de mon analyse de performances. Le serveur de production est dimensionné pour supporter la charge de plusieurs dizaines, voire centaines d'utilisateurs connectés simultanément. Les performances du serveur sont donc capables d'absorber le choc et de fournir des performances optimales lorsque nécessaire.

Les environnements de développement, eux, ne comptent qu'une petite dizaine d'utilisateurs connectés simultanément tout au plus. Les serveurs sont donc dimensionnés de manière à délivrer des performances raisonnables en essayant de les garder les plus petits possible. De cette sorte, il devient possible de démultiplier les environnements de développement sur les mêmes systèmes pour les différents projets de l'entreprise et ainsi réduire les coûts.

C'est en me rappelant cela que j'ai réalisé que les tests ont été réalisés sur des environnements inégaux au niveau des performances. La migration ayant déjà été validée, je vais relancer un test de performance sur la nouvelle base pour valider les performances réelles obtenues une fois la migration terminée.

La mise en production étant un environnement sensible et n'ayant pas les compétences d'administrateur système nécessaires, je n'ai pas pu participer à la mise en place de la mise en production des changements. J'ai cependant pu vérifier que tout fonctionnait comme attendu en inspectant l'environnement de production.

Une fois le serveur de production mis à jour avec les nouveaux éléments, j'effectue un nouveau test de performance pour valider que le biais de différence de performance entre les environnements est bien la source du problème. Je conserve les données obtenues précédemment pour la base PostgreSQL 14 et je récupère le résultat de l'outil de benchmark une fois exécuté sur la nouvelle base et sur le même environnement pour une comparaison correcte.

	Postgres_14	**Postgres_16**	**Gap_From_14_To_16**
Total_Io_Count	323527	346761	23234
Total_Duration	3577961	4025135	−447174
Identical_Ios_Duration	3577257	3980590	403333
Average_Ios_Load_Duration	11,05923462	10,82812255	0,54857306

Dans le tableau Tableau 3 les résultats bruts ne donnent pas forcément de gros indices sur les différences par rapport à la première analyse comportant potentiellement un biais. Je constate une inversion sur la quantité de données traitées, cette fois-ci la nouvelle base de données à eu plus d'OIs à traiter et par conséquent les temps de traitements ont aussi augmenté. Une comparaison avancée me permet d'obtenir un comparatif plus explicite entre les deux bases.

Identical_Group_Count	3044
Identical_Ios_Perf_Gain	10,13%
Total_Perf_Gain	11,11%

Les résultats du tableau Tableau 4 confirment ma supposition. La différence entre les deux environnements a biaisé les résultats des analyses. Sur les OIs identiques, on retrouve cette fois-ci les dix pourcents annoncés par l'équipe de développement de PostgreSQL. Encore mieux, je remarque que pour les OIs analysés au global dans les deux bases, la différence de performances grimpe jusqu'à onze pourcents.

Le projet bénéficie donc des améliorations de performances promises par l'équipe PostgreSQL, mais surtout, de performances encore plus accrues que prévues par rapport à l'ancien système.

3.4.2. Problèmes rencontrés

Après la mise en place du nouveau système, une erreur de compatibilité dans les transactions entre l'application Observo et InfSuite est apparue. Malgré les nombreuses étapes de tests, les anaylses préliminaires, la mise en place du système sur un serveur intermédiaire nommé staging qui est très similaire à l'environnement de production, le problème n'a pas pu être détécté avant. Il n'est survenu que lors de la mise en place des mises à jour sur le serveur de production.

Le problème concernait une erreur de compatibilité avec les fameuses date lors de l'échange de données entre les deux application. La différence de format fournie par InfSuite n'était pas valide côté Observo, le système retrouvait donc la transaction en échec.

3.5. Ouverture

Au cours des trois années d'alternance passées au sein de l'entreprise Unit Solution, j'ai activement pu participer au développement et à la maintenance du projet InfSuite. Comme pour l'année dernière, j'ai apporté des améliorations à l'outil d'administration. Toujours orienté sur la gestion de la base de données, j'ai pu créer une fonctionnalité pour vérifier les scripts exécutés précédemment sur la base de données. Comme expliqué plus tôt dans ce document, InfSuite utilise des scripts SQL pour permettre de mettre à jour les schéma de base de données et ainsi éviter des modifications manuelles en base de données. Pour garder une liste des scripts déjà exécutés sur le serveur, un écran dans l'application permet de voir rapidement l'historique d'exécution des scripts.

Ajouter ce genre de gestion de base de données dans l'application permet également de remettre en question la manière actuelle de faire les choses pour tenter d'améliorer le processus.
Actuellement, pour gérer les schémas de base de données, nous utilisons le logiciel Power Designer. Il permet de gérer les définitions des modèles de données avec les relations, les définitions de clés primaires, etc. Il ne permet cependant que de générer des définitions, il ne permet pas d'effectuer les actions sur la base de données, il reste totalement hors connexion.
Une fois les modifications effectuées dans l'outil, il génère le nouveau schéma de base avec les définitions de tables, colonnes,... Pour créer les scripts qui vont mettre à jour la base de données avec ces nouvelles définitions, il faut aller chercher dans le fichier de schéma de base de données, modifié par Power Designer, les différentes modifications. On peut alors créer un nouveau fichier `.sql` dans lequel on va y ajouter les instructions SQL permettant d'effectuer ces modifications. Une fois le script rédigé, on peut tester que tout fonctionne correctement sur une base de développement. Si tout fonctionne comme prévu, alors il faut le dupliquer dans les dossiers de configuration spécifiques à chaque environnement et pousser toutes ces modifications sur Azure Devops.

Toute cette gestion complexifie la simple de tâche de mettre à jour les informations en base de données. Il serait plus convenant de trouver une alternative moins complexe pour effectuer cette tâche. Après avoir mis à jour la base de données, on peut alors légitimement se poser la question :Comment simplifier l'administration d'une base de données.
On pourrait alors partir par exemple sur une implémentation dans l'outil d'administration du projet pour effectuer cette tâche longue et récurrente.

Conclusion

Pour conclure cette seconde année de master en alternance chez Unit Solutions, travailler sur un sujet étant aussi ancré dans le projet qu'est la base de données est d'une importance cruciale pour mon parcours académique et professionnel. Travailler sur la migration de base de données pour l'application InfSuite m'a permis de développer et d'approfondir des compétences essentielles en gestion de données et en technologies web.

Cette tâche complexe, nécessitant une compréhension approfondie de l'architecture logicielles et de la gestion de données (syntaxe SQL, compréhension d'un SGBDR, etc.), m'a offert dans un premier temps une experience pratique précieuse, et dans un second temps, l'occasion de renforcer ma capacité à interpréter et résoudre une problématique technique complexe et sensible.

L'opportunité d'avoir un rôle aussi marqué sur ce projet a renforcé ma confiance en mes compétences professionnelles et m'a permis de contribuer de manière tangible à cette mission a fort intérêt pour l'entreprise. Le soutien constant et les conseils de mon chef de projet, des encadrants de la formation, ainsi que l'environnement de travail collaboratif chez Unit Solutions, ont été un facteur clé de la réussite de cette mission.

Le travail réalisé tout au long de cette année pour la migration de base de données est maintenant installé en production et fonctionne de manière nominale pour tous les utilisateurs et services tiers de l'application. Cette migration permettra à l'avenir d'étendre les fonctionnalités et les performances de l'application.

Cette alternance, marquée par des défis techniques aussi bien sur l'apprentissage de nouveaux éléments que leur mise en pratique, a non seulement enrichi mon parcours académique, mais a également posé les bases solides pour mon avenir professionnel. En effet, cette expérience se concrétise par mon intégration prochaine en CDI au sein de l'entreprise, marquant ainsi le début de ma carrière dans le domaine professionnel à temps plein. Ce parcours chez Unit Solutions, alliant théorie et pratique, a été une étape déterminante pour mon développement personnel et professionnel, et je me réjouis de poursuivre cette collaboration fructueuse.

Glossaire

- **Back-End** : Some content
- **Dump** : Un fichier « dump » est un fichier de sauvegarde qui contient une copie de toutes les données et métadonnées d'une base de données à un instant T.
- **EF** : Entity Framework qui se decline également en Entity Framework Core, est une librairie C# qui sert d'ORM
- **Front-End** : Also some content
- **OFROU** : L'Office Fédérale des Routes est l'autorité suisse compétente pour l'infrastructure routière
- **ORM** : De l'anglais Object Relational Mapping, permet d'interfacer les object code avec les données contenues en base de données.
- **SGBDR** : Un Système de gestion de base de données relationnelle est le système composant une base de données.
- **Sharding** : Allié à la réplication, il permet de répartir la charge sur plusieurs instances d'un même serveur.

Liste des abréviations

- **EF** : Entity Framework
- **OFROU** : Office Fédérale des Routes
- **ORM** : Object Relational Mapping
- **SGBDR** : Système de gestion de base de données relationnelle

Bibliographie et webographie

Table des matières

Annexes

Résumé

Abstract

Mots-clés

-
-

Julien Ivars