Implémenter un système de sauvegarde dans ObservoAdmin

Rapport d'alternance

Master Informatique et Mobilité

Tom Willemin
Promotion : 2022/2023

Tuteur professionnel : M. Vellicus Thibault

Tuteur pédagogique : M. Elbaz Mounir

Organisme d'accueil : Unit Solutions AG

Implémenter un système de sauvegarde dans ObservoAdmin

Tom Willemin

Remerciements

Je tiens à remercier toutes les personnes qui ont contribué au succès de mon alternance, de mes années d'études et qui m'ont aidé lors de la rédaction de ce mémoire.

Je voudrais dans un premier temps remercier le directeur de l'entreprise Unit Solutions, M. Thierry Moebel de son accueil et pour m'avoir poussé à poursuive mes études tout en m'accordant une place dans l'entreprise.

Je remercie mon tuteur d'alternance, M. Thibault Vellicus pour sa confiance accordée en m'attribuant la responsabilité de tâches importantes comme celle présentée dans ce mémoire. Je le remercie également pour m'avoir confié des missions afin de me faire monter en compétence sur les différentes applications et technologies de l'entreprise. Je souhaite également remercier M. Thomas Fritsch, étudiant alternant de l'UHA 4.0 qui est aussi employé chez Unit Solutions, pour m'avoir épaulé tout au long de l'année.

J'aimerais remercier l'entièreté des employés de Unit Solutions pour la convivialité instaurée dans l'entreprise, permettant de travailler plus confortablement.

Je remercie également toute l'équipe pédagogique de l'école UHA 4.0, M. Florent Bourgeois, M. Daniel Da Fonseca et M. Pierre Schuller qui ont été les enseignants chargés de mon suivi. Ils étaient toujours disponibles en cas de difficulté ou de blocage dans le projet. Mme. Audrey Brunsperger en tant que chargée d'animation a également répondu présente tout au long de l'année lorsque j'avais besoin de précieux renseignements.

Et pour finir, j'aimerais remercier ma famille qui m'a soutenu tout au long de mes années d'études et lors de mon alternance. Merci également à eux pour leurs précieux conseils sur la rédaction de ce mémoire.

Sommaire

Remerciements 3
Sommaire 4
Introduction 4
1 Projets à l'UHA 4.0 5
1.1 Contextualisation 5
1.2 UHA 4.0.4 5
2 L'entreprise Unit Solutions 6
2.1 Présentation de l'entreprise 6
2.2 Organisation et méthodologie de travail 7
3 Présentation d'Observo et ObservoAdmin 8
4 Gestion de sauvegarde 9
4.1 Problématique 9
4.2 Préparation du projet ObservoAdmin 10
4.2.1 Contexte 10
4.2.2 Détermination des composants à refactoriser 12
4.2.3 Refactorisation des composants 13
4.2.4 Utilisation de services 16
4.2.5 Limitations rencontrées 16
4.3 Implémentation de la sauvegarde 17
4.3.1 Détecter les modifications 17
4.3.2 Création d'éléments 18
4.3.3 Implémentation de la sauvegarde avec la gestion des erreurs 20
4.3.4 Détecter la navigation 24
4.3.5 Problèmes rencontrés 26
5 Conclusion 27
L'apport de l'alternance à l'entreprise 27
Résumé 28
Mots-clés 28
Glossaire 29
Sources 32
Annexes 33

Introduction

Suite à l'obtention de mon Baccalauréat technologique “Sciences et Technologies de l’Industrie et du Développement Durable”, ayant une appétence pour les nouvelles technologies informatiques j’ai décidé d’intégrer l’école UHA 4.0. Mes trois années d’études au sein de cette école m’ont permis d’acquérir les compétences nécessaires pour valider et obtenir le diplôme de Licence développeur informatique. Cela m’a donné l’opportunité ensuite de suivre le parcours « Master Informatique et Mobilité » toujours au sein de l’école UHA 4.0.

Au cours de ma dernière année de préparation à la Licence, j’ai effectué une alternance dans l’entreprise Unit Solutions. Étant donné que l’entreprise en a été satisfaite, cela m’a ouvert la porte à un renouvellement de mon contrat dans le cadre de mon DU 4.0.4. Elle a été répartie en 3 mois d’école et 9 mois d’entreprise.

Unit Solutions a été mon premier choix pour effectuer mon alternance car elle utilise des technologies modernes, développe dans des domaines qui m’intéressent (web et mobile), possède beaucoup de projets techniquement intéressants et a un cadre de travail conviviale.

Sur place, j'ai été intégré à l'équipe de développement des projets nommés Observo et ObservoAdmin. Mon alternance a eu deux objectifs. Le premier est d’apporter de nouvelles fonctionnalités à l’application mobile Observo. Et le deuxième est de compléter les fonctionnalités de l’application web ObservoAdmin afin de remplacer l’application d’administration existante. En effet la solution d’administration permettant de gérer l’environnement d’Observo utilisé actuellement fonctionne avec l’outil Access. Avec le temps, de plus en plus de fonctionnalités y ont été ajoutées ce qui ralentit son fonctionnement et le rend instable. Ce qui justifie la nécessité de le remplacer.

Dans ce mémoire, je vais commencer par présenter mon parcours effectué au sein de l’école UHA 4.0, ensuite je vais présenter l’organisme d'accueil en détaillant ses différents projets avec leur problématique. Et pour finir je vais me concentrer sur une fonctionnalité développée durant l’alternance. Ici, la fonctionnalité est un système de sauvegarde au sein de l’application ObservoAdmin.

1 Projets à l'UHA 4.0

1.1 Contextualisation

Une année d'étude dans l'école UHA 4.0 du parcours « Master Informatique et Mobilité » est découpée en deux parties. Dans la première, sur une durée de trois mois, nous effectuons ce que nous appelons un fil rouge. C'est un projet informatique regroupant des objectifs nous permettant d'acquérir et de mettre en pratique les connaissances nécessaires à la validation du Diplôme Universitaire. De plus, nous suivons des topos qui sont assurés par des professeurs et intervenants, en lien avec les connaissances requises au fil rouge. Ensuite dans la deuxième partie, nous suivons une période de neuf mois en alternance au sein d'une entreprise que je détail dans ce rapport.

1.2 UHA 4.0.4

Durant mon année d'étude en DU 4.0.4, j'ai pu aborder des notions en Intelligence Artificiel, en fouille de données, en sécurité informatique et en algorithmie. Cela comprend de l'algorithmie géométrique (qui a pu ensuite être appliqué à des images) et de l'optimisation combinatoire. Ce qui m'a permis en autre, de compresser des images ou encore de trouver une meilleure solution parmi un nombre proche de l'infini de choix.

Le sujet du fil rouge a été l'automatisation d'un jardin afin de le rendre autonome. Le système devait préconiser le bien-être de la plante tout en optimisant sa consommation en eau. Pour cela nous avons disposé de 3 plantes, équipées d'un oya permettant leur irrigation avec des caméras et plusieurs capteurs. Afin de relever l'humidité et la température du milieu de la plante.

Je me suis concentré sur la détection du manque d'irrigation de la plante. Parmi les solutions explorées, la plus intéressante a été l'utilisation de la forme de ses feuilles. En effet, lorsque l'espèce de la plante utilisée dans le fil rouge (Calathéa Médaillon) s'assèche, ses feuilles se courbent. En utilisant une caméra récupérant une image de celles-ci toutes les 5 minutes. J'ai appliqué un algorithme de segmentation à l'image en utilisant la méthode d'Otsu (Cf. glossaire). Une fois la feuille détectée, je lui ai appliqué automatiquement un calcul de rondeur. Ce qui m'a permis de définir l'état de la plante en fonction de l'évolution de la rondeur de ses feuilles.

Le seuil de rondeur permettant de définir l'état de la plante et donc de classifier son état a été choisi arbitrairement. Avec plus de temps, il aurait été intéressant de mettre en place un arbre de décision. Un arbre de décision est un algorithme de Machine Learning (Cf. glossaire) permettant dans notre situation, de prédire un seuil optimal en fonction de l'assèchement de la feuille.

Et pour finir mon année d'étude en DU 4.0.4, j'ai suivi une alternance au sein de l'entreprise Unit Solutions que je présente dans ce rapport.

2 L'entreprise Unit Solutions

2.1 Présentation de l'entreprise

Unit Solutions est une entreprise Suisse, située à Allschwil dans le canton de Bâle-Campagne. Elle a été fondée en 1986 initialement sous le nom de « CAD Rechenzentrum AG », nom qui a été changé en 2015 pour « Unit Solutions AG ».

L'entreprise est spécialisée dans la conception d'applications modernes web et mobiles, dans les systèmes de localisation (Système d'information géographique SIG, plans, modèles 3D, photos, GPS) et dans l'exploitation de données et la génération de rapports.

Unit Solutions comptabilise une vingtaine d'employés décomposés en plusieurs équipes, avec pour chacune d'entre elles, un responsable (Cf. annexe 1). L'équipe du support est directement en relation avec les clients, ils sont chargés du bon fonctionnement des outils utilisés en interne à l'entreprise et des serveurs utilisés pour la publication des différentes solutions développées. Ensuite chaque équipe a la responsabilité du suivi et du développement d'une ou plusieurs applications. J'ai été intégré dans l'équipe de développement Mobility-team (Cf. la figure 1, ci-après).

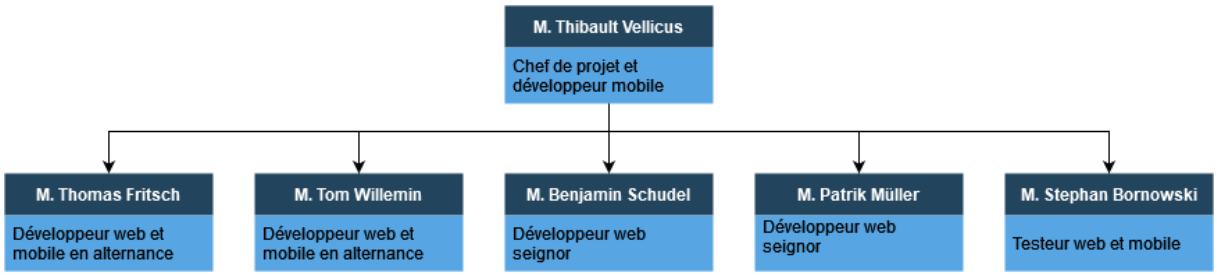


Figure 1 : Organigramme de l'équipe de développement Mobility-team

L'entreprise concentre principalement son activité sur la réalisation des projets InfKuba, Kuba, Mobilité douce et Observo :

- InfKuba et Kuba sont une suite de logiciels permettant le suivi et la maintenance d'infrastructures tels qu'une route, un pont, une barrière anti-avalanche ou encore un filet anti-éboulement.
- Observo est une application mobile permettant de relever des observations sur des infrastructures au travers d'une carte. Initialement autonome, elle tend à collaborer avec la suite InfKuba. En effet, nous pouvons désormais importer et exporter des données entre les applications. L'utilisation d'Observo est plutôt destinée à une utilisation en déplacement en se rendant directement auprès de l'infrastructure ciblée.
- Mobilité douce est une application web, facilitant la coordination à travers les cantons de Suisse pour l'entretien de chemins, la planification d'itinéraires ainsi que la signalisation pour les modes de mobilité douce comme la randonnée, le vélo et le roller.

2.2 Organisation et méthodologie de travail

Au sein de l'équipe Mobility-team, notre méthode de travail s'approche de la méthodologie Agile (Cf. glossaire). Cela se traduit par une daily meeting (Cf. glossaire) tous les matins nous permettant de faire un point sur le travail effectué et les objectifs de la journée. Cela nous permet de savoir où en est chacun, de discuter en cas de blocage ou de partager nos points de vue. Tous les lundis nous faisons une « Montag-Zitsung », nous présentons à l'entièreté de l'entreprise ce que nous avons effectué la semaine précédente et ce que nous allons faire la semaine à venir. Cela nous permet d'avoir une dynamique de travail toutes les semaines et de savoir ce que font les autres employés de l'entreprise.

Afin de coordonner notre travail nous utilisons l'outil Jira. Les demandes des clients et de M. Thierry Moebel sont traduites en Epic et en User story (Cf. glossaire) par M. Thibault Vellicus. Ensuite nous intégrons les fonctionnalités et corrigeons les bogues en fonction des tickets (Cf. glossaire) se trouvant dans le Sprint actif (Cf. glossaire). Au cours du Sprint, les tâches sont validées par M. Stephan Bornowski qui est responsable de vérifier le bon fonctionnement de l'application. Cela nous permet lors de l'ajout d'une fonctionnalité, d'éviter des régressions.

Dans l'environnement d'Observo, chaque fonctionnalité majeure donne lieu à un Sprint. À la fin de celui-ci, nous pratiquons une série de Smoke tests (Cf. glossaire) visant les modifications récemment ajoutées. En résultat de ces Smoke tests, nous dressons une liste de bogues à corriger en fonction de leur degré d'urgence. Ensuite nous effectuons une Sprint retrospective (Cf. glossaire) permettant à chacun de s'exprimer sur les points qui se sont bien déroulés et les points à améliorer afin de pouvoir les corriger dans les futurs Sprints. Une fois l'application publiée nous ouvrons un nouveau Sprint, ce qui nous permet d'entretenir une dynamique de publication.

Afin de centraliser le travail de toute l'équipe sur le même projet, nous utilisons l'outil de versionning Git d'Azure Devops (Cf. glossaire). Nous utilisons également Azure Devops afin de compiler (Cf. glossaire) l'application et de lancer l'entièreté de ses tests unitaires (Cf. glossaire) avant la validation d'une Pull Request (Cf. glossaire). Pour simplifier la gestion du Git nous utilisons le client (Cf. glossaire) graphique, SourceTree.

Durant la résolution d'une tâche dans Jira, nous suivons le flow de travail schématisé dans la figure 2 ci-après. Dans un premier temps, les développeurs (M. Thomas Fritsch et moi-même) sont chargés de traiter les tâches (colonne 1) en fonction de leur priorité. Une fois effectuée (colonne 2), nous soumettons une PR qui doit être validée par un développeur différent que celui chargé de la tâche (colonne 3). Ce qui permet de trouver des oublis, des dysfonctionnements ou des mauvaises pratiques au sein du code. Une fois la PR validée, le développeur va passer la tâche dans la colonne 4. C'est au chef de projet (M. Thibault Vellicus) de valider le déploiement de la tâche sur un environnement de test accessible au sein de l'entreprise (colonne 5). Ensuite c'est au testeur (M. Stephan Bornowski) de valider la tâche en fonction du bon comportement de l'application (colonne 7). Si la tâche n'est pas complétée, alors elle sera passée dans la colonne 6. Et pour finir, le développeur responsable de tâche se l'attribue dans la colonne 2 pour suivre à nouveau le même flow jusqu'à sa validation.

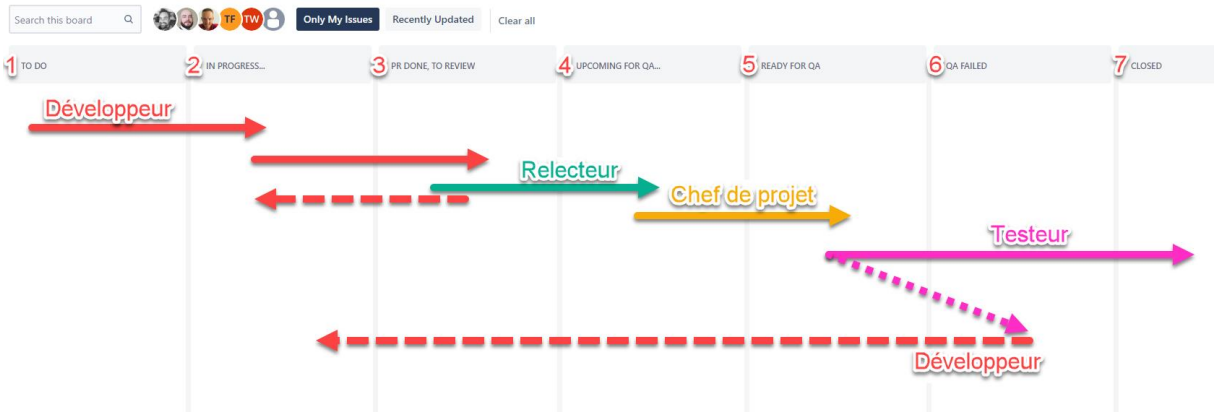


Figure 2 : Schéma du flow de travail suivi par l'équipe de développement Mobility-team

Au cours de l'année nous avions la possibilité de passer 40% de notre temps en télétravail. Pour cela l'équipe du support a mis en place des VPN et un logiciel nous permettant de prendre possession à distance de nos postes de travail dans l'entreprise.

3 Présentation d'Observo et ObservoAdmin

L'objectif de l'application mobile Observo est de permettre à ses utilisateurs de mener des inspections auprès des infrastructures concernées.

Elle répond à la demande de numérisation et donc de simplification des processus d'inspections d'ouvrages. Par exemple, la Suisse a un budget assigné à chacun de ses Cantons pour l'entretien et le suivi d'ouvrages. Avant Observo, toutes les démarches se faisaient sur papier et il n'y avait pas d'historique disponible pour un ouvrage.

L'application est utilisée en Suisse, en France et en Allemagne. Elle compte plus de 40 clients pour plus de 290 utilisateurs.

Observo est composée d'une carte interactive permettant d'afficher et naviguer au travers d'ouvrages. Ces ouvrages sont représentés par des icônes ou des formes personnalisables (Cf. annexe 2). L'application contient un système de formulaire dynamique permettant à l'utilisateur d'inspecter et d'enregistrer ses données comme l'état global d'une structure ou la description d'une fissure (Cf. annexe 3) en fonction de l'infrastructure ciblée. Elle contient également un système de génération de rapports à partir de ses inspections qui permettent à l'utilisateur d'avoir un rendu de celles-ci.

L'objectif de l'application web ObservoAdmin est de remplacer le système d'administration Access déjà en place depuis le lancement du projet Observo. Access est un logiciel faisant partie de la suite Microsoft Office, permettant de créer des vues rapidement et simplement qui modifient directement le contenu d'une Base De Données. La problématique est qu'avec le temps, de plus en plus de fonctionnalités sont ajoutées dans Access ce qui le rend de moins en moins maintenable. En effet, l'application rencontre des ralentissements et des bogues. De plus, il n'y a pas de logique métier exécutée sur un serveur entre la vue et le modèle (Cf. glossaire), ce qui complique l'ajout des fonctionnalités comme l'envoi de mails par exemple.

ObservoAdmin permet donc d'administrer l'environnement d'Observo. Cela comprend la modification d'utilisateurs, de groupes d'utilisateurs, de mandants (Cf. glossaire), de listes, de

formulaires, de formules ect. Elle est principalement utilisée afin de configurer des formulaires pour des clients ou pour résoudre des problèmes survenus auprès des clients comme la perte de données ou l'inaccessibilité d'informations sur l'application mobile.

Le projet ObservoAdmin a été créé en 2020 et est actuellement utilisé en production (Cf. glossaire). Il ne contient pas encore toutes les fonctionnalités nécessaires, permettant le remplacement total de la solution Access. Il est utilisé par les employés de l'entreprise Unit Solutions, afin de gérer le contenu d'Observo. Par la suite, ObservoAdmin pourrait être utilisé par les différents clients de Unit Solutions afin de leur donner plus d'autonomie.

Le projet a été développé avec Angular (Cf. glossaire) pour le client, Asp .Net Core (Cf. glossaire) pour le backend (Cf. glossaire) avec l'utilisation du Framework (Cf. glossaire) Hangfire (Cf. glossaire) et SQL Server pour la BDD (Cf. glossaire).

4 Gestion de sauvegarde

4.1 Problématique

Au sein de l'outil Access, il est difficilement possible de perdre des modifications. En effet l'application étant directement reliée à la Base De Données, les modifications sont enregistrées après la navigation.

Dans ObservoAdmin nous n'avions pas de système de sauvegarde uniforme et sécurisé. Sous les formulaires nous avions un bouton permettant de sauvegarder les modifications. Dans les grilles, la sauvegarde se faisait au travers d'un bouton intégré au composant. Pour l'ajout d'un nouvel élément, cela se faisait au travers d'une modale contenant un formulaire. Si nous naviguons dans l'application, nous perdons toutes les modifications effectuées. Étant donné qu'une modification peut représenter beaucoup de travail, c'est un risque de perte trop élevé. Ce problème a été relevé par l'équipe du Support et mon responsable et est devenu une tâche prioritaire.

Nous avons besoin d'un système empêchant toutes pertes de modifications par l'utilisateur. Pour cela, comme décrit dans la figure 3 ci-après, nous devons détecter les modifications effectuées. Une fois détectées, un bouton permettant de sauvegarder doit s'activer. Si l'utilisateur navigue, une modal bloquant la navigation doit permettre à l'utilisateur de revenir en arrière, continuer de naviguer en sauvegardant ou de continuer de naviguer en ignorant les modifications. Cela doit être appliqué lors de la modification d'un formulaire, d'une grille ou durant la création d'un nouvel élément. La suppression d'un élément doit être directement appliquée suite à la confirmation de l'utilisateur via une modale.

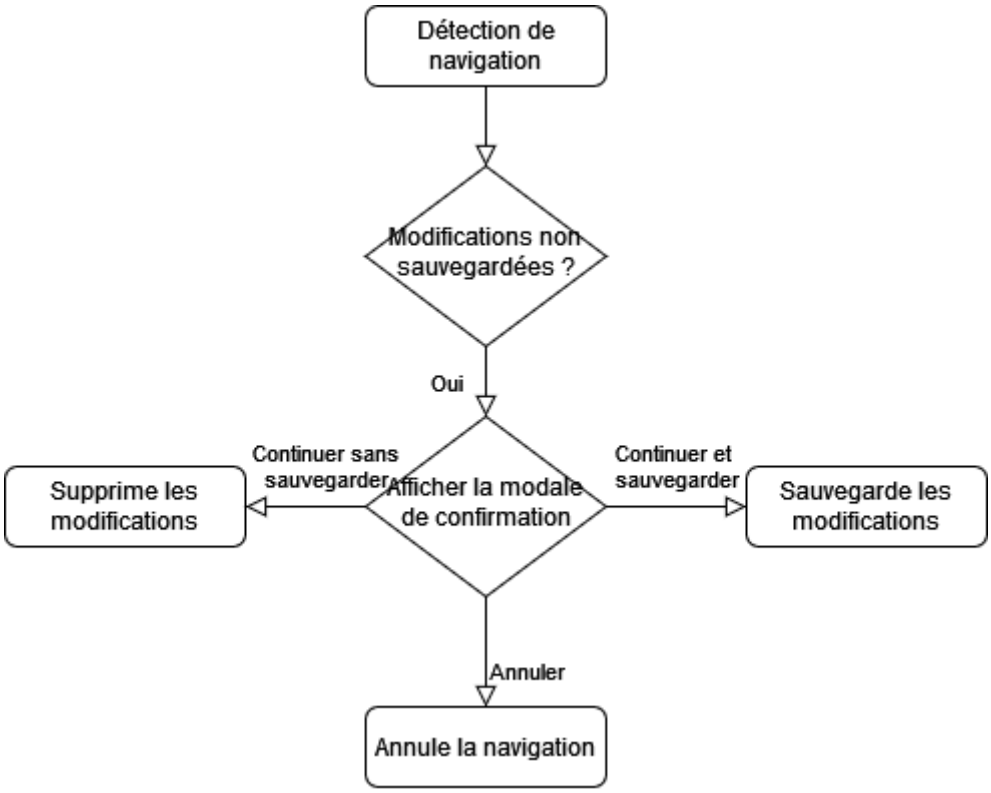


Figure 3 : Diagramme représentant le fonctionnement de la modale de confirmation de navigation

Les IHM (Cf. Glossaire) liées à l'édition doivent être identiques d'une page à une autre afin de rendre son utilisation plus simple. De plus, il m'a été demandé d'uniformiser les interfaces avec les fonctions de création et de suppression d'éléments. Mon objectif a également été d'intégrer ces fonctions dans les pages de l'application web où elles sont manquantes.

Pour l'intégration de cette fonctionnalité j'ai pu compter sur mon collègue alternant de l'UHA 4.0 M. Thomas Fritsch, sur M. Benjamin Schudel et M. Patrik Müller qui sont deux développeurs confirmés de l'entreprise et enfin sur M. Stephan Bornowski qui a effectué des Smoke tests me permettant de trouver les bogues non corrigés.

4.2 Préparation du projet ObservoAdmin

4.2.1 Contexte

Les besoins du projet ObservoAdmin n'ont pas été définis depuis sa création. Ce qui a contribué à la mise en place d'une architecture de projet ne correspondant pas à ses besoins. En effet, avec le nombre grandissant de fonctionnalités intégrées au projet, il devient de plus en plus coûteux d'en ajouter une nouvelle en restant compatible avec le reste de l'écosystème.

Cela a également été dû à une demande de développement qui a été de produire une application fonctionnelle, avec peu de budget et le plus rapidement possible. D'après la théorie du « Triangle de fer » émise par le Dr. Martin Barnes en 1969, la gestion de projet se positionne sur 3 axes. Le temps, le coût et la qualité du résultat. En suivant cette théorie, le projet ObservoAdmin ne peut pas répondre à toutes ces attentes. Ce qui a été répercuté sur la qualité

du code. On peut retrouver dans le projet par exemple de la répétition de code, des services (Cf. glossaire) ou fonctions à plusieurs responsabilités, du code peu typé, peu segmenté et l'occultation (Cf. glossaire) des fonctions ou attributs peu définis. Dans le langage Typescript, si nous ne définissons pas la visibilité d'une fonction et que la Classe (Cf. Glossaire) est publique, alors la fonction sera par défaut publique. Avec ces mauvaises pratiques, comme l'a expliqué M. Robert C. Martin dans le chapitre 1 du livre Clean Architecture, l'intégration de nouvelles fonctionnalités devient de plus en plus coûteuse.

Pour intégrer le système de gestion de sauvegarde et pour pouvoir garder une bonne productivité au cours de la maturation du projet ObservoAdmin, j'ai décidé de procéder à la refactorisation des éléments clés du projet.

Pour cela j'ai essayé au maximum de suivre les principes de développement informatique KISS et SOLID.

Le principe KISS pour « Keep It Simple Stupide » ou « Garde ça simple, idiot » en français est une ligne directrice qui préconise la simplicité dans la conception de fonctionnalité. Toute complexité non indispensable devrait être évitée. Cela permet d'entreprendre l'implémentation de fonctionnalités avec le minimum de code et de complexité pour compléter la MVP (Cf. Glossaire). Afin d'éviter de se retrouver dans le cas où le développeur complique son code afin de le rendre utilisable en prévision de futurs cas de figures. Etant donné que ces cas de figures peuvent changer ou peuvent ne jamais être nécessaires, du temps de développement a été perdu, la lisibilité du code est impactée et du code inutilisé ou inutile, alourdit le projet.

Le principe SOLID a pour objectif d'amener le développeur à une production d'architecture logicielle dans le cadre de la Programmation Orientée Objet plus compréhensible, flexible et maintenable. C'est un acronyme ou la lettre « S » a la signification « Single Responsibility Principle » (SRP) ou bien « Principe de responsabilité unique » en français. Ce qui désigne, qu'une Classe doit avoir uniquement une seule responsabilité et donc elle ne doit changer que pour une seule raison. La lettre « O » pour « Open-Closed Principle » (OCP) ou « Principe ouvert/fermé » en français, qualifie la nécessité qu'un élément de code doit être ouvert aux nouvelles fonctionnalités mais doit également être fermé aux modifications. La lettre « L » pour « Liskov Substitution Principle » (LSP) ou « Principe de substitution de Liskov » en français, se rapproche de la programmation par contrat (Cf. Glossaire). C'est-à-dire qu'à l'échelle d'un programme orienté objet, le code qui utilise des sous-types tels que des implémentations d'une interface, ne doit pas avoir un comportement différent en fonction du sous-type utilisé. Ce principe (comme les autres) s'applique également à l'échelle de l'architecture logiciel. Par exemple, nous pouvons avoir un groupe de service qui doivent tous répondre à la même REST interface. La lettre « I » pour « Interface Segregation Principle » (ISP) ou « Principe de ségrégation des interfaces » en français, signifie qu'il faut éviter de dépendre de code dont on n'a pas besoin en le découpant dans de plus petites interfaces (Cf. glossaire). Cela permet d'éviter de devoir recompiler son programme à cause de modifications dans du code non utilisé. Et enfin, la lettre « D » pour « Dependency Inversion Principle » (DIP) ou « Principe d'inversion de dépendance » en français, désigne que le code doit dépendre au maximum d'abstractions plutôt que d'implémentations concrètes. Car une modification à une abstraction va toujours amener à la modification de son implémentation alors que le sens inverse ne s'applique pas toujours. Ce qui rend le code qui utilise des abstractions, plus stable.

4.2.2 Détermination des composants à refactoriser

Avant de pouvoir déterminer les composants à refactoriser, il a fallu définir le code dupliqué. En effet, d'après M. Robert C. Martin « il existe deux types de duplications. Il y a la vraie duplication, dans laquelle chaque copie apportée à une instance nécessite la même modification pour chaque copie de cette instance. Il y a ensuite la duplication fausse ou accidentelle. Si deux sections de code apparemment dupliquées évoluent selon des voies différentes, si elles changent à des rythmes différents et pour des raisons différentes l'une de l'autre ». En se basant sur cette observation au sein de l'application ObservoAdmin, j'ai estimé que le code dupliqué peut être refactorisé et regroupé sous trois principaux types de composants réutilisables.

Le premier est le composant qui va contenir la page. Neuf pages dans ObservoAdmin nécessitants le système de sauvegarde sont composés des mêmes éléments principaux d'affichage. En effet, l'architecture de la page type (voir la figure 4 ci-dessous) est composée sur la partie de gauche, d'une grille de navigation permettant de sélectionner un élément qui sera affiché sur la partie de droite. La partie de droite permet d'afficher les informations sur cet élément pour ensuite les modifier au travers de différents onglets. Ces pages ont également la responsabilité de charger les données, afin de les distribuer dans ses différents sous-composants. En revanche, elles ne possèdent aucun socle commun. Ce qui pose un problème d'uniformité dans le fonctionnement des pages de l'application. En effet, certaines parties de code nécessaires dans chacune des pages sont dupliquées. Avec le temps et au cours d'implémentation de nouvelles fonctionnalités, ce code « commun » est modifié dans une page mais pas dans le reste de l'application. Ce qui provoque des comportements différents entre ces pages, alors que le fonctionnement attendu doit-être le même.

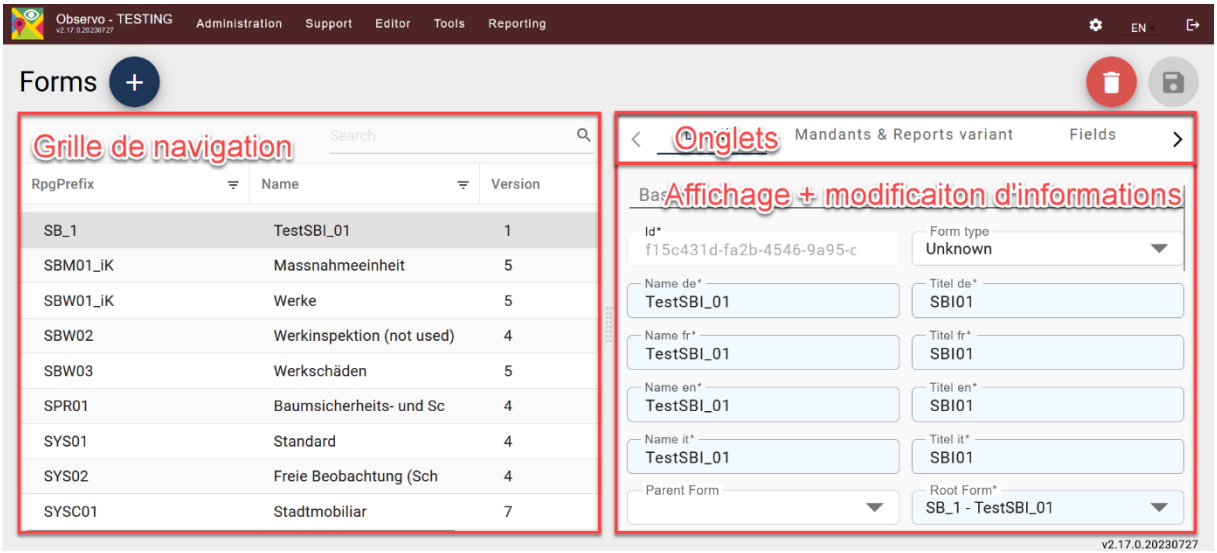


Figure 4 : Capture d'écran annotée des composants principaux d'une page type dans ObservoAdmin.

Le second composant regroupe les formulaires. ObservoAdmin utilise plus de 15 formulaires différents. Ils peuvent se trouver en mode de lecture seule ou en mode d'édition. Jusqu'à présent, les formulaires étaient directement implémentés dans le composant de la page. Nous n'utilisions pas de balise « <form> » permettant de regrouper tous les champs du même formulaire ensemble. Ces champs étaient implémentés au travers de 4 composants réutilisables

développé en interne à l'entreprise. Le premier était utilisé pour toutes les valeurs transitant sous forme de String (Cf. Glossaire) ou de type Date. Il regroupait les champs texte libre, texte area (texte libre permettant d'écrire un plus grand volume de texte) et date. Le second était utilisé pour les valeurs de type Boolean (Cf. Glossaire) pour afficher des checkboxs. Et les deux derniers étaient utilisés pour les listes déroulantes avec une ou plusieurs sélections. Cette méthode ne respecte pas la lettre S du principe SOLID. En effet, comme l'a cité M. Robert C. Martin, « un module devrait être responsable d'un, et un seul, acteur ». En revanche, ici un module est responsable de plusieurs acteurs (différents types de champs). Ce qui peut amener à un conflit de fonctionnalité. Par exemple, lorsque l'on remplit le champ date, il faut caster (Cf. glossaire) le String reçu en type Date, en revanche pour le champ texte il ne faut pas caster le String. Ces nombreux conflits nous contraignent à ajouter de plus en plus de conditions sur le type de champ actuellement utilisé ce qui réduit la maintenabilité et la lisibilité du code.

Enfin, le troisième composant regroupe les grilles. Durant ma précédente année d'alternance au sein de l'entreprise, j'ai eu la responsabilité d'intégrer un composant permettant d'afficher des grilles. Ce composant est une surcouche entre l'élément de la libraire Syncfusion nommé « Grid » et notre application. Il permettait de configurer la « Grid » en fonction de nos besoins et des paramètres que nous lui donnons lors de son implémentation. La grille permet donc d'afficher des données au travers d'un tableau. Ce tableau peut être édité à la manière d'Excel. Un grand nombre de fonctionnalités optionnelles gravitent ensuite autour. Ces fonctionnalités sont par exemple, la possibilité de trier/renommer/ordonner les colonnes du tableau, définir le type de valeur qui se trouvera dans la colonne (l'édition des cellules du tableau change en fonction de son type) ou encore changer le mode de sélection des données (cela peut être une sélection cellule par cellule ou une sélection ligne par ligne). Ce composant de grille est donc le composant majeur d'ObservoAdmin afin d'afficher et d'éditer des données. Il est utilisé 16 fois. Cela s'explique par le fait qu'ObservoAdmin est une application d'administration permettant la modification et l'affichage de grand volume de données. L'affichage sous forme de tableau correspond tout à fait à ce besoin. Le problème est que, tout comme les composants de champs réutilisables de formulaires, la lettre « S » du principe SOLID n'est pas respectée. En effet, « Un composant doit changer pour une et une seule raison ». Or, dans notre cas, le composant pourrait être modifié pour une fonctionnalité spécifique au mode de sélection ligne par ligne et spécifique au mode cellule par cellule. Ce qui pose un problème si le fonctionnement de la sélection ligne par ligne impacte le fonctionnement de la sélection cellule par cellule.

4.2.3 Refactorisation des composants

Afin de centraliser la logique commune à chaque implémentation du même type de composant, nous avons 3 possibilités différentes. À noter que dans chacune des implémentations, le respect du principe DIP est de vigueur. Le composant dépend au maximum d'abstractions et d'interfaces plutôt que de classes concrètes. Cela à l'exception d'utilisations de modules concrets mais stables. Par exemple, l'objet Date est une classe concrète mais pour des raisons de facilité, il est utilisé sans inversion de dépendance. Le respect du principe DIP est d'autant plus important du fait de la réutilisation de composants sur une grande étendue de l'application. En effet, nous souhaitons que le code réutilisé soit le plus stable et donc le moins susceptible d'être changé.

La première possibilité, consiste à centraliser l'entièreté de la logique dite métier (Cf. Glossaire) et d'affichage dans un même composant. Celui-ci sera réutilisable là où son implémentation est

nécessaire (voir la figure 5 ci-dessous). Pour le développeur, ce composant est en quelque sorte une boîte noire. Ce qui veut dire qu'il n'a pas besoin de se soucier du fonctionnement interne au composant pour l'utiliser. Il doit en revanche prendre connaissance de son interface. Cette méthode comporte néanmoins les désavantages suivants. Le non-respect du principe SRP car le composant sera responsable de différents acteurs (l'affichage a un comportement différent en fonction de l'implémentation) ce qui lui donnera une raison de changer pour un acteur, puis pour un second et nous amènera à un conflit de fonctionnalité. De plus, il sera plus difficile de respecter le principe OCP. Car le composant est responsable de l'affichage qui est le plus propice à changer. Ce qui est en désaccord avec le principe.

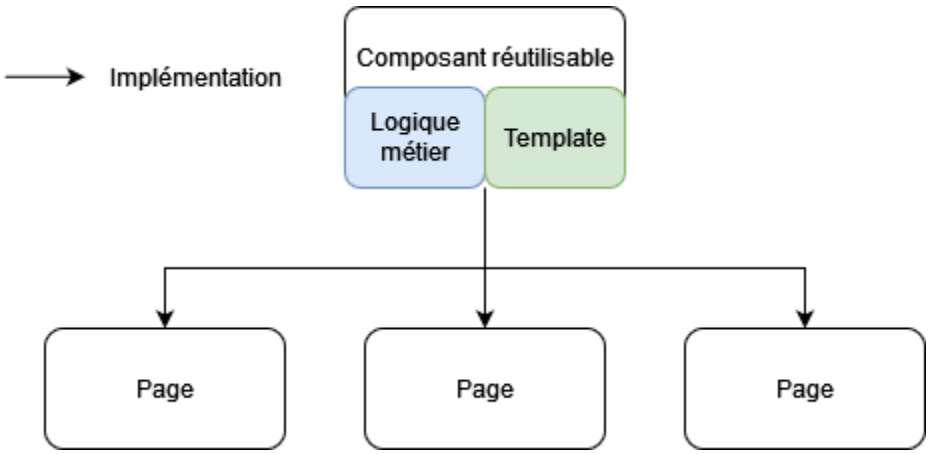


Figure 5 : Représentation de l'architecture du composant réutilisable avec un composant central.

La seconde possibilité est de centraliser uniquement la logique métier commune à chacune des utilisations comme schématisé dans la figure 6 ci-après. Plus concrètement, le composant est caractérisé par une classe abstraite contenant la logique commune. Il sera implémenté là où l'utilisation du composant est nécessaire. C'est au développeur de définir le template (Cf. Glossaire) du composant pour chacune de ses utilisations. Cette méthode a comme avantage d'avoir plus de souplesse. Car un affichage peut posséder des spécificités en fonction de son utilisation qui n'ont donc pas besoin d'être pris en compte par la centralisation de la logique. Cela respecte les principes OCP et SRP. OCP car la partie la plus propice à changer, ici l'affichage, est implémentée dans chacune des utilisations du composant, ce qui réduit les chances d'avoir le code commun modifié. SRP car le composant n'est responsable que d'un acteur. Ici le fonctionnement de la logique métier nécessaire dans chacune de ses implémentations. En revanche, le fait de définir l'affichage pour chaque utilisation du composant, alourdit le code source du projet.

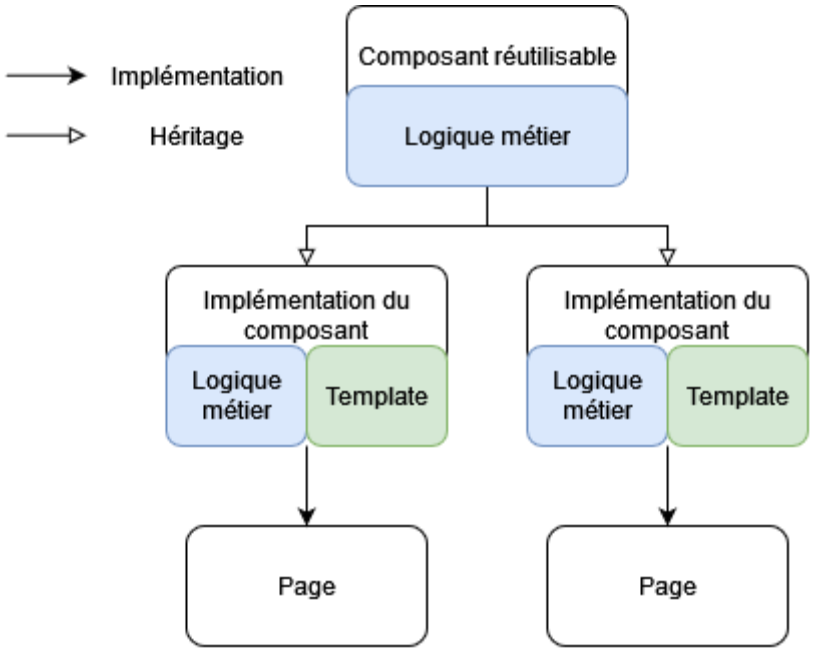


Figure 6 : Représentation de l'architecture du composant réutilisable avec une implémentation par utilisation.

Et pour finir, la troisième possibilité consiste à concentrer la logique commune dans un service ou un helper (Cf. glossaire) qui sera utilisé par le développeur afin d'implémenter la logique métier à l'affichage. Cela permet tout comme la seconde possibilité, d'abstraire la logique du composant de la vue (Cf. glossaire). En revanche, en comparaison avec l'utilisation d'une classe abstraite ou une interface, le développeur ne sera pas contraint d'implémenter toutes les fonctions se trouvant dans le service ou le helper. Ce qui peut être nécessaire pour certaines méthodes (Cf. glossaire) requises au bon fonctionnement du composant.

Étant donné que l'implémentation au travers d'un composant a été la solution utilisée pour la centralisation des grilles et que cela a posé des problèmes de complexité et de maintenabilité, j'ai écarté cette possibilité. L'utilisation des services et de helpers a comme limitation de ne pas contraindre le développeur à définir les fonctions centralisant la logique. Nous ne pouvons pas partir du principe que les fonctions seront utilisées si elles ne sont pas imposées. C'est pourquoi j'ai décidé d'utiliser des classes abstraites. Cela rend les intégrations plus robustes et plus souples. En effet, les classes abstraites laissent également ouvert la possibilité, à l'avenir, de créer des spécialisations de ces composants en utilisant l'héritage. Dans le cas où un composant peut être regroupé en plusieurs sous-composants permettant de répondre aux besoins d'utilisations multiples du même cas d'usage. Par exemple, si nous avons besoin dans 5 grilles différentes d'un même mode d'édition, nous pourrons créer une classe abstraite héritant de la principale et incluant la logique supplémentaire. De ce fait, ce procédé rend l'implémentation souple.

4.2.4 Utilisation de services

Avec l'utilisation de composants nous avons pu découper le code source des pages en plusieurs fichiers. Le problème est que jusqu'à présent, l'échange de données au sein de la même page se faisait au travers de variables globales (Cf. Glossaire). Cette pratique peut être poursuivie en utilisant la fonction du framework Angular « @Input » et « @Output ». Ce sont des décorateurs de variables permettant l'échange d'information entre un composant parent et ses composants enfants. En revanche, il est recommandé d'utiliser un service lorsque la même information est nécessaire dans plusieurs composants à la fois. Cela évite de répéter les déclarations de « @Input » et « @Output » et cela rend l'information disponible dans les composants ne se trouvant pas dans la hiérarchie parent/enfants. Un service dans le framework Angular est une classe contenant des méthodes et des attributs utilisables dans le reste de l'application. Cette classe suit par défaut le design pattern Singleton et est utilisable grâce à l'injection de dépendance. Le principe du « Singleton » est de rendre accessible une même instance d'une classe dans plusieurs fichiers. Afin de partager une information au travers d'un service, nous allons principalement utiliser l'Observer pattern. Il consiste à écouter les évènements de modification d'une variable afin d'obtenir la dernière version de celle-ci et d'effectuer des actions en fonction.

Dans l'objectif de structurer le partage d'information, afin d'être réutilisé dans l'entièreté de l'application et de rendre la recherche d'une information plus facile. J'ai décidé de répartir un service par page et par onglet dans le cas où celui-ci nécessite plusieurs données. Cela permet également d'avoir des fichiers de code les plus courts que possible afin d'en garder une bonne lisibilité.

Afin de garder plus de cohérence dans le projet, j'ai décidé d'attribuer au composant de page, la responsabilité de chargement des données dans ses différents services. Cela permet de trouver plus facilement les appels au serveur et d'éviter les risques de duplication de code car les appels sont centralisés. De plus, cela réduit les comportements dissemblables en fonction de l'emplacement où se trouve le chargement des données. En effet, les composants ne chargent pas tous en même temps et donc ont besoin des données à des moments différés.

4.2.5 Limitations rencontrées

La refactorisation de l'application d'administration ObservoAdmin a été soumise à plusieurs limitations. L'objectif principal de cette initiative a été de rendre l'application plus prédictible, maintenable et évolutive. Néanmoins les contraintes ci-dessous ont influencé le processus de refactorisation.

L'une des limitations les plus significatives a été le facteur de temps. La refactorisation d'une application de la taille d'ObservoAdmin nécessite de l'analyse et de la modification sur une longue période. Cependant, la limite des parties à refactoriser est difficilement identifiable. De ce fait, il a été nécessaire de restreindre la portée de la refactorisation aux principaux axes de l'application, ici les formulaires, les grilles et les pages.

Un autre facteur limitant a été le budget attribué à la refactorisation. En effet, ObservoAdmin est une plateforme d'administration facilitant la gestion de l'environnement d'Observo. Etant donné que ce n'est pas le projet principal de l'entreprise, le budget disponible en est proportionnel. De plus, la refactorisation n'apporte pas de nouvelle fonctionnalité. D'un point

de vue utilisation, excepté un gain de stabilité, le travail effectué n'est pas notable. Ce qui a eu un impact sur le budget.

Et pour finir, la tâche de refactorisation n'était pas l'objectif initial de la demande. Elle a été entreprise dans le but de faciliter l'intégration d'un système de sauvegarde présenté dans la suite de ce rapport. La refactorisation d'une application est un processus continu et itératif. De nouvelles opportunités d'amélioration peuvent émerger à mesure que l'application continue de se développer et d'évoluer.

4.3 Implémentation de la sauvegarde

4.3.1 Détecter les modifications

Dans le cadre du système de sauvegarde nous allons nous concentrer dans ce chapitre sur la partie permettant de détecter les modifications. Cela va nous permettre par la suite, d'effectuer diverses actions en fonction des interactions de l'utilisateur.

Pour réaliser cette fonctionnalité, j'ai exploité la centralisation des formulaires et des grilles qui a été mise en place lors de la refactorisation de l'application (décrite dans le chapitre 4.2.3 « Refactorisation des composants »). J'ai également créé un service de traque de modifications, avec comme mission d'enregistrer tous les éléments modifiés.

Étant donné qu'une modification peut se faire uniquement au travers d'un composant de grille ou de formulaire, le service doit contenir une représentation des composants modifiés. Pour cela j'ai ajouté une liste de clé/valeur avec comme clé le nom du composant, et la valeur un boolean représentant l'état de modification de celui-ci. Le fait d'utiliser comme clé, le nom du composant limite la détection de modification à une seule instance par composant. Ce qui veut dire que nous ne pouvons pas utiliser le même composant de grille ou de formulaire plusieurs fois. Cette limitation peut être évitée par la génération d'un identifiant unique par instance de composant. En revanche, cela complexifierait le débogage (Cf. glossaire). En effet, le nom du composant est un nom clair et facile de compréhension. De plus, comme décrit dans le chapitre 4.2.3 « Refactorisation des composants », une implémentation concrète d'un composant doit être effectuée pour chaque instance de celui-ci. Afin de suivre les principes SOLID et donc de garder le code le plus souple que possible. Ce qui retire la possibilité d'avoir deux composants avec le même nom.

Avant la refactorisation, afin de détecter les modifications effectuées dans un formulaire nous aurions dû écouter les modifications sur chacun des champs au travers du Observer Pattern. Etant donné qu'ObservoAdmin comporte plus de 15 formulaires dont chacun peut compter jusqu'à une vingtaine de champs cela demanderait trop de code dupliqué et trop de ressources au navigateur chargeant la page.

Désormais, en passant par des formulaires nous pouvons écouter un évènement s'activant à chaque modification de celui-ci. De plus, grâce au socle commun à tous les formulaires (la classe abstraite) nous pouvons centraliser l'écoute de l'évènement à un seul endroit. Cela réduit considérablement le code nécessaire et augmente sa lisibilité. Cela permet également d'avoir le même comportement auprès de tous les composants de formulaire. Ce qui est nécessaire sur un fonctionnement critique comme la détection d'une modification. Mais encore, grâce à l'utilisation de formulaire, nous pouvons centraliser la détection d'une annulation de modification. C'est-à-dire, quand l'utilisateur inverse ses modifications et se retrouve dans

l'état initial du formulaire. Dans ce cas, l'état de modification du composant doit-être annulé. Pour ce faire, lorsque le formulaire est rempli avec les données provenant du serveur, je les dupliques et les garde dans une variable. Lorsqu'une modification est détectée, je compare le contenu du formulaire aux données initiales. Si les deux données sont égales, alors nous pouvons déduire que les modifications effectuées par l'utilisateur ont été annulées. Et nous pouvons donc envoyer au service que le composant n'est plus modifié.

De même que pour les composants de formulaires, les composants de grilles possèdent un socle commun. Ce qui nous permet également d'écouter et de transmettre l'état d'édition d'une grille, auprès du service qui les centralise.

Grâce à cette centralisation, nous pouvons activer ou désactiver le bouton de sauvegarde et à l'inverse pour le bouton de suppression. Ils se trouvent dans chacune des pages de l'application ObservoAdmin afin de valider une modification ou une suppression d'élément. Le tout en fonction de l'état d'édition. Cela permet d'informer l'utilisateur de l'état dans lequel il se trouve et d'empêcher la suppression lorsque des modifications sont en cours.

Figure 7 : Capture d'écran des boutons de validation d'édition et de suppression d'ObservoAdmin dans l'état sans, puis avec modification en cours.

4.3.2 Création d'éléments

Au sein de l'application ObservoAdmin, de nombreuses pages représentent des éléments comme des utilisateurs, des groupes ou encore des mandants. Ces pages ont comme objectif de permettre la modification et la création de ces éléments. Nous allons donc dans ce chapitre, nous concentrer sur la fonctionnalité de création de nouveaux éléments.

Afin de permettre à l'utilisateur de créer de nouveaux éléments nous avons deux possibilités.

La première et la plus simple, consiste à afficher une modale contenant un formulaire. Ce formulaire représente l'objet à créer. Une fois rempli et validé, l'élément est généré.

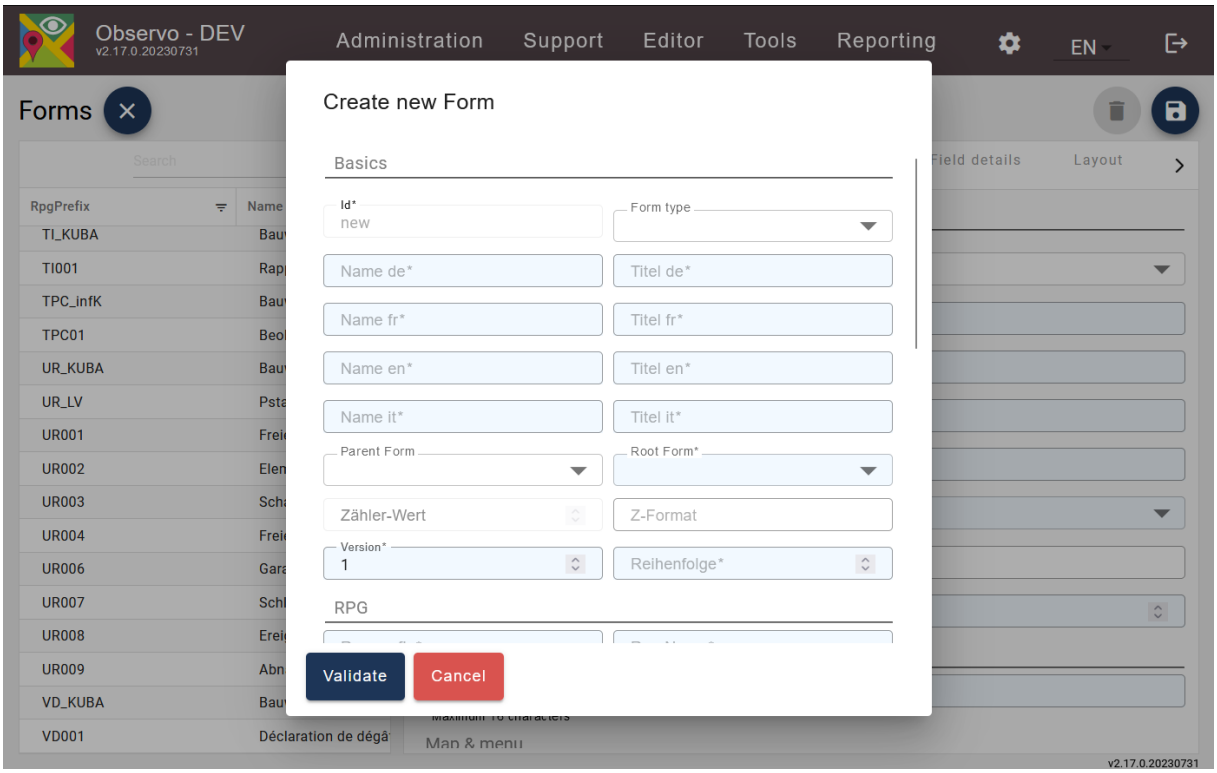


Figure 8 : Capture d'écran d'ObservoAdmin avec la création d'élément au travers d'une modale.

La seconde possibilité consiste à vider le formulaire permettant normalement de modifier l'élément sélectionné dans la page. Une fois vide, l'utilisateur peut le remplir et le valider. Ce qui entraîne la création de l'élément.

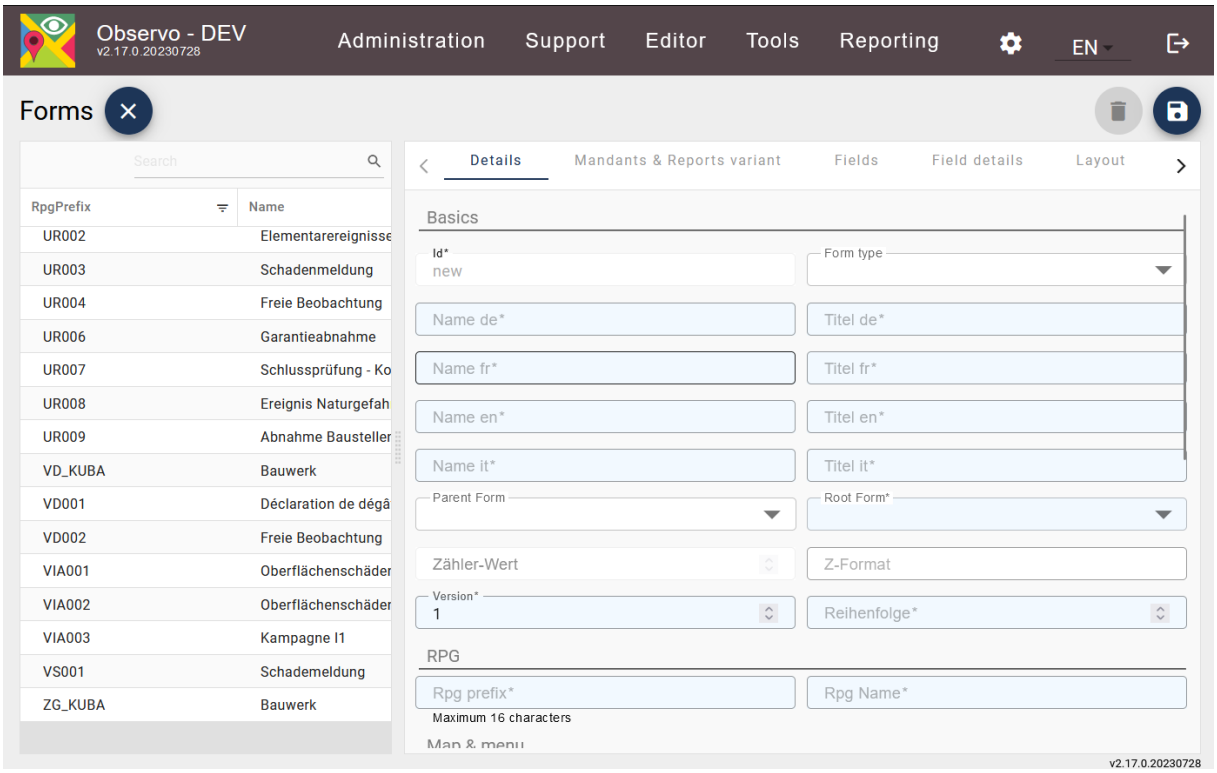


Figure 9 : Capture d'écran d'ObservoAdmin avec la création d'élément au travers du formulaire d'édition.

Afin de définir l'interface utilisateur la plus simple à être utilisée, j'ai mis en place les deux possibilités dans des pages différentes. Une fois la nouvelle version de l'application publiée, j'ai recueilli les retours des différents utilisateurs. La comparaison a pu être possible dans des pages différentes du fait de leur ressemblance structurelle. Le résultat des retours nous permet de déterminer la seconde possibilité (sans l'utilisation de modale) comme la plus adéquate. Cela s'explique par le fait que la navigation n'est pas bloquée par une modale. Ce qui la rend plus fluide.

L'information sur l'état de création de la page, tout comme son état de modification doit être centralisé dans le service permettant la traque des modifications. En effet, les composants se trouvant dans la page ont également besoin d'avoir accès à l'état de création. Par exemple, lors de la création, le formulaire principal (un sous composant à la page) doit se vider.

Enfin, lorsque la page passe en mode de création, le bouton permettant de sauvegarder doit s'activer afin de permettre à l'utilisateur de valider la création d'un élément.

4.3.3 Implémentation de la sauvegarde avec la gestion des erreurs

Suite à la détection de modification et l'ajout de la possibilité de créer et modifier des éléments, nous avons besoin de lier ces éditions avec les fonctions de sauvegarde (ici une fonction de sauvegarde est une fonction contenant la logique permettant de sauvegarder des modifications). Cela dans l'objectif d'envoyer les modifications au serveur lorsque l'utilisateur clique sur le bouton sauvegarder. Dans ce chapitre nous allons donc aborder la logique permettant de

sauvegarder les modifications en utilisant le service de traque de modification (vu dans les chapitres plus haut).

Les fonctions de sauvegarde contiennent de la logique spécifique à chacune des intégrations. C'est pourquoi, ces fonctions ne peuvent être centralisées. J'ai décidé de les positionner dans le composant principal au lieu de les placer dans chacun des composants de grille et de formulaires qui leur sont respectif. Car cela simplifie leur recherche au sein des fichiers du code source du projet. De plus, nous évitons d'avoir des fichiers trop grands grâce à la répartition de ces fonctions au travers des différents composants de page. Nous allons définir une fonction de sauvegarde par composant d'édition (formulaire ou grille). De cette manière nous allons pouvoir sauvegarder uniquement les composants comportant une modification. La segmentation de la sauvegarde sera utile lorsque l'on souhaitera gérer les erreurs rencontrées du côté du serveur.

Une fois ces fonctions de sauvegarde définies dans les composants de page, nous avons besoin de les enregistrer auprès du service de traque de modification. Afin de les exécuter en fonction des composants détectés comme modifiés. De la même manière que la centralisation de l'état de modification des composants (expliqué dans le chapitre 4.3.1 « Détecter les modifications »), à l'initialisation de l'application, nous allons les sauvegarder sous forme de clé/valeur (Cf. la figure 10 ci-dessous). La clé est le nom du composant d'édition auquel la sauvegarde est rattachée. La valeur correspond à un callback. La fonction de callback est une fonction passée dans une autre fonction en tant qu'argument afin d'être appelé. Dans notre cas, ce callback contient la fonction de sauvegarde se trouvant dans le composant de page.

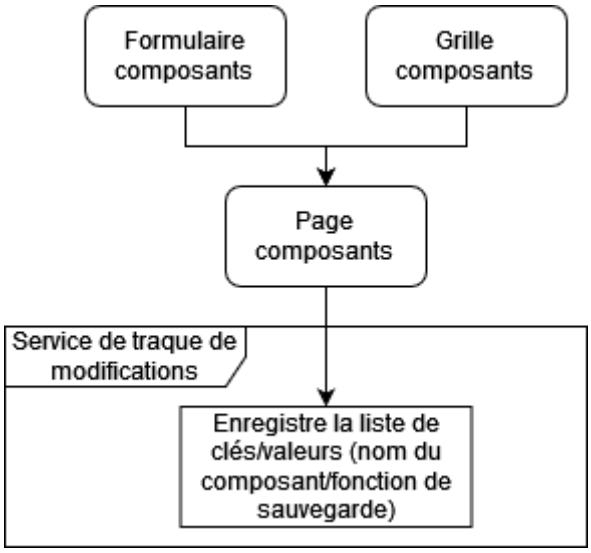


Figure 10 : Schéma de l'enregistrement des fonctions de sauvegarde auprès du service de traque de modifications.

De cette manière, lorsque l'utilisateur clique sur le bouton de sauvegarde, au sein du service de traque de modifications, nous pouvons récupérer le nom des tous les composants ayant une modification détectée. Ensuite nous pouvons appeler la fonction de sauvegarde se trouvant dans le callback correspondant au nom du composant. Et pour finir, nous pouvons remettre à zéro la liste de composants comportant des modifications comme décrit dans la figure 11 ci-après.

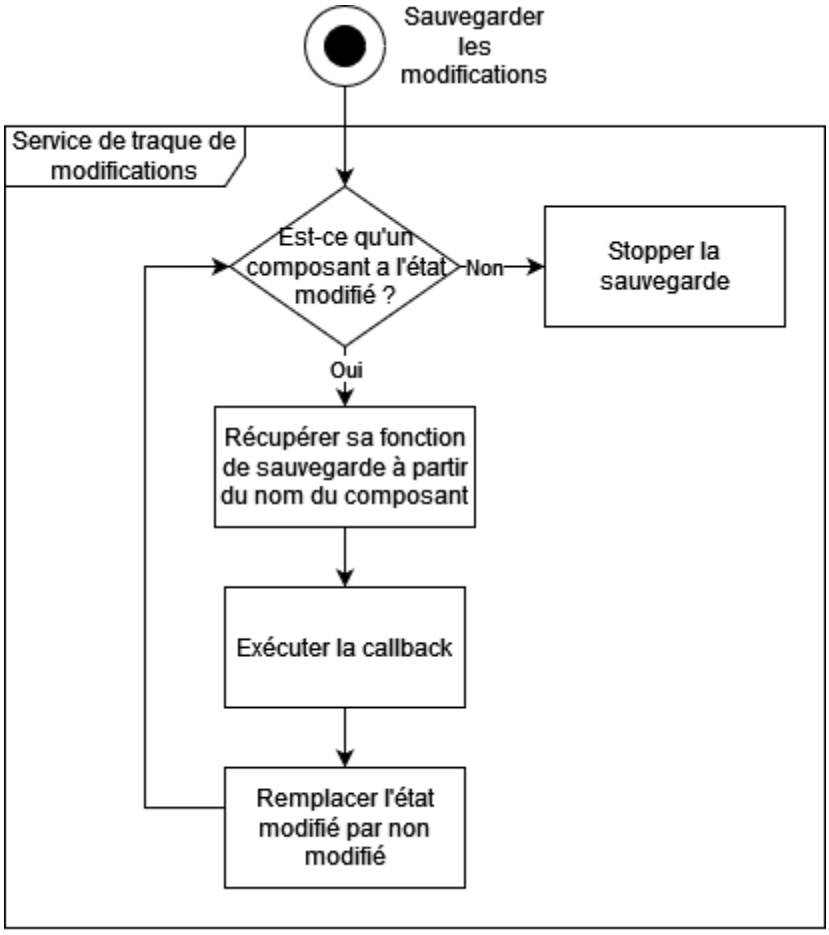


Figure 11 : Schéma de l'exécution de la sauvegarde au sein du service de traque de modifications.

Tout comme la sauvegarde des composants, l'annulation des modifications doit être possible. En effet, l'utilisateur doit être capable d'annuler ses modifications effectuées par mégarde. Pour cela des fonctions nommées discard sont définies dans les composants d'édition. Contrairement aux fonctions de sauvegarde, elles ne sont pas définies dans les composants de page. Car la logique nécessaire à l'annulation des modifications est propre au type de composant d'édition.

Dans les implémentations des formulaires il faut définir une fonction permettant de remplacer les valeurs affichées avec les valeurs initiales. En effet, comme expliqué dans le chapitre 4.3.1 « Détecter les modifications », lors du chargement du formulaire j'enregistre une version des valeurs dans une constante dans l'optique de garder une version sans modifications. Lors de l'appel de la fonction de discard, ces valeurs sont donc appliquées au formulaire.

Dans les implémentations des grilles j'ai choisi d'envoyer une nouvelle requête au serveur afin de remplacer les valeurs affichées contenant des modifications par des données correspondantes à la BDD. Ce choix a été guidé par la nécessité d'avoir la version des valeurs la plus à jour dans les grilles. Une grille peut comporter plusieurs milliers de valeurs. Elles sont donc plus susceptibles d'être changées par un autre utilisateur qu'une dizaine de valeurs se trouvant dans un formulaire.

Durant l'exécution de la sauvegarde, des erreurs peuvent survenir tel que des champs manquants dans des formulaires, des mauvaises valeurs (du texte à la place d'un nombre par exemple) dans les grilles ou encore une erreur provenant du serveur. Le comportement attendu de la sauvegarde lors de la rencontre d'erreur, est son annulation complète. En effet, nous ne voulons pas sauvegarder la moitié d'une modification. Une telle sauvegarde rendrait les données non prédictibles, ce qui est l'inverse souhaité. En revanche, les modifications doivent être gardées en mémoire dans le navigateur, laissant la possibilité à l'utilisateur de corriger les erreurs. De plus, si un bogue survient, cela permettrait à l'utilisateur de sauvegarder ses modifications dans un tableau Excel par exemple pour les grilles le temps d'une correction.

Pour arriver à ces fins, dans le service de traque de modifications, j'ai choisi de stocker les callbacks de sauvegarde sous forme de Promises. Une Promise est un objet utilisé pour réaliser des traitements asynchrones. Elle représente une valeur qui peut être disponible à un instant variable voir jamais (source). L'objet peut se trouver dans trois états différents. En attente qui est l'état initial. Tenue qui est atteint lorsque que la Promise est complète et qu'elle a retourné la valeur promise. Et rompue, lorsque l'opération a échoué. C'est ce dernier état qui nous est utile afin de savoir quand une erreur survient lors de la sauvegarde. Suite à cela, nous arrêtons l'exécution des callbacks au sein du service de traque de modifications comme décrit dans la figure 12 ci-dessous. Nous en profitons pour afficher un message de confirmation ou d'erreur pour chacun des composants concernés par la sauvegarde afin de tenir l'utilisateur informé.

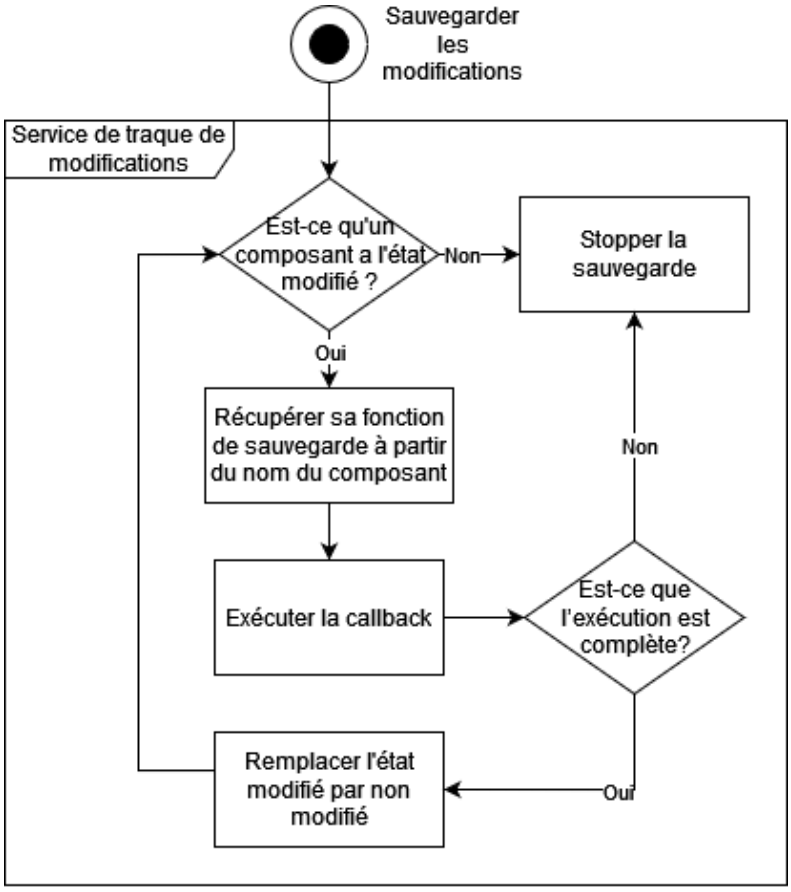


Figure 12 : Schéma de l'exécution de la sauvegarde avec la gestion d'erreur au sein du service de traque de modifications.

Plus haut, j'ai mentionné la nécessité de sauvegarder les modifications par morceaux. Ici la sauvegarde se fait par composant. En revanche, lorsqu'une erreur est rencontrée la sauvegarde du composant concerné doit être annulée. Pour les composants de formulaire, cela ne nécessite pas de précautions particulières du côté du serveur. Car la plupart du temps les modifications concernent une entité particulière et donc la requête à la BDD est faite en une transaction. Pour les composants de grille, une modification peut concerner plusieurs entités, mais est surtout composée d'une première requête de mise à jour, d'une seconde de modification et d'une troisième de suppression. Si la requête de modification échoue, les deux autres ne doivent pas s'exécuter. Il est nécessaire dans ce cas, d'agencer la fonction du serveur afin de sauvegarder l'ensemble des modifications en une seule transaction SQL. De cette manière, une erreur annule l'intégralité des modifications.

4.3.4 Détecter la navigation

Lorsque l'utilisateur a effectué des modifications et souhaite naviguer, il doit avoir le choix sur le traitement des modifications dans l'objectif de ne pas en perdre par mégarde. Dans ce chapitre nous allons nous concentrer sur cette partie du système de sauvegarde.

L'utilisateur doit avoir trois choix lors de la navigation avec des modifications non sauvegardées. Il doit pouvoir valider ses modifications (les sauvegarder) et continuer la navigation, annuler ses modifications (les perdres) et continuer la navigation ou bien garder ses modifications et annuler la navigation (Cf. la figure 3 dans le chapitre 4.1 « Problématique »).

Pour proposer un choix à l'utilisateur nous devons savoir à quel moment l'afficher. La navigation peut être faite à trois endroits différents.

Le premier, lorsque l'utilisateur navigue au travers de la grille d'éléments se trouvant sur la gauche, les informations sur la partie droite sont mises à jour. Si ces informations comportent des modifications, elles seront perdues.

Le second endroit, lorsque l'utilisateur passe d'un onglet à un autre sur la partie de droite. Étant donné que ces onglets affichent des informations sur le même élément (sélectionné dans la grille de gauche), il arrive qu'une information soit partagée dans plusieurs onglets. Une modification dans un onglet doit donc également se trouver dans le reste des onglets. C'est pourquoi il est plus simple de contraindre l'utilisateur de sauvegarder les modifications au changement d'onglet.

Et enfin, lorsque l'utilisateur navigue au travers des menus se trouvant dans l'entête de l'application web.

La grille émet un évènement lorsque l'utilisateur sélectionne un élément. Grâce à celui-ci nous savons alors quand l'utilisateur souhaite naviguer d'un élément à un autre et donc d'afficher la modale. Néanmoins la grille comporte une limitation. Lorsque l'utilisateur sélectionne un élément, nous ne pouvons pas arrêter la sélection. C'est pourquoi j'ai dû sauvegarder l'ancien élément sélectionné jusqu'à la réponse de l'utilisateur. Si la navigation est annulée, nous pouvons sélectionner l'ancien élément.

De la même manière que pour les grilles, nous utilisons l'évènement qu'émettent les onglets lors de leur sélection. Tout comme les grilles, nous ne pouvons pas arrêter la sélection jusqu'à

ce que l'utilisateur fasse son choix. Nous devons donc garder en mémoire l'ancien onglet sélectionné.

La navigation au travers de menus est différente des autres modes de navigation. Nous aurions pu détecter les cliques dans les menus pour ensuite afficher la modale de confirmation. En revanche, si dans le future nous décidons d'ajouter un nouveau moyen de navigation (en plus des menus), l'affichage de la modale ne se fera plus. C'est pourquoi j'ai utilisé la fonctionnalité de Guard (Cf. glossaire) dans Angular. Les Guards sont des règles pouvant être appliquées sur des routes (des chemins d'accès à des pages) dans une application Angular. Plus précisément j'ai décidé d'utiliser le Guard « canDeactivate » permettant d'annuler une navigation si la règle définie n'est pas respectée. L'avantage est que l'annulation de la navigation se fait avant qu'elle ait commencé. Ce qui nous permet de garder intactes toutes les modifications dans la page actuelle.

Lorsque nous avons détecté une navigation, nous allons le transmettre au service de traque de modification. De ce fait, il va pouvoir, à partir des informations que les composants d'édition lui ont transmis, lancer la sauvegarde ou non des modifications (Cf. la figure 13 ci-dessous).

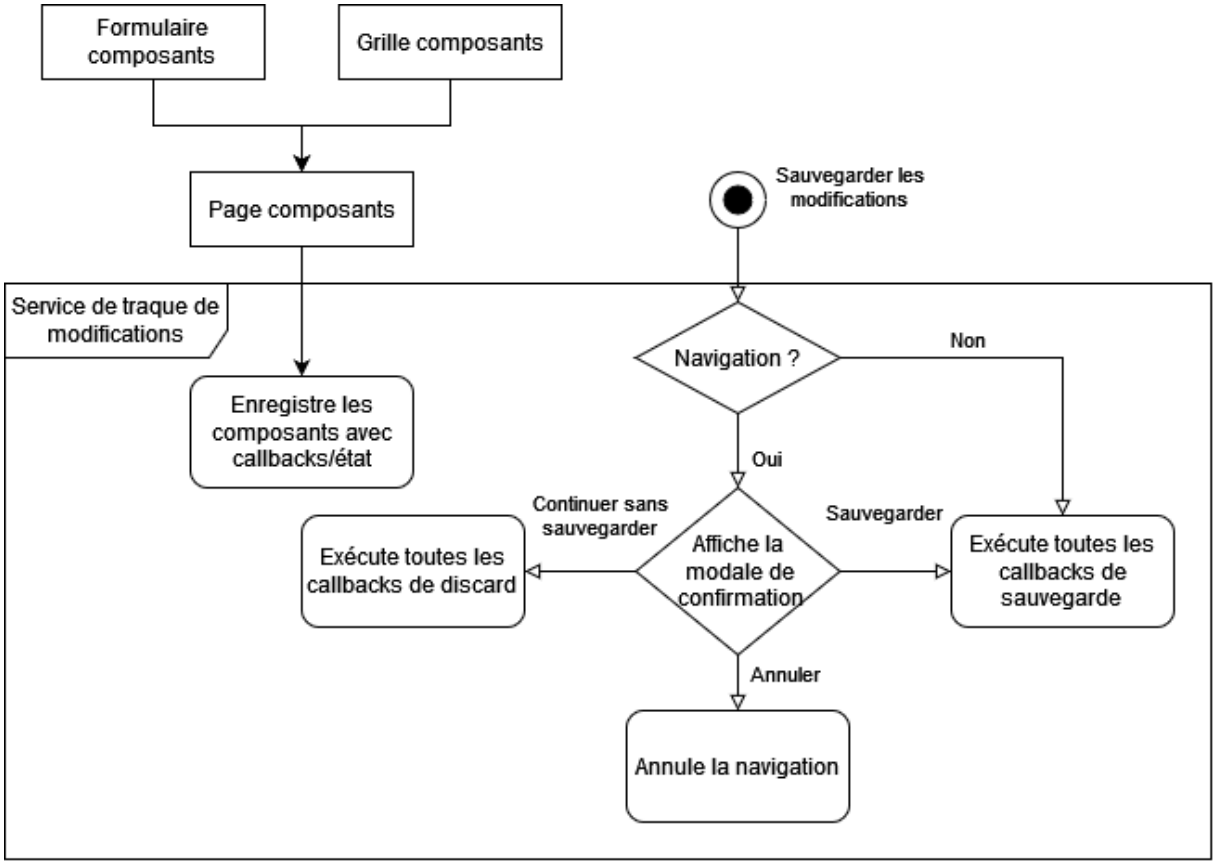


Figure 13 : Schéma de l'enregistrement des composants et de l'exécution de la sauvegarde ou de l'annulation en fonction de la navigation au sein du service de traque de modifications.

4.3.5 Problèmes rencontrés

Après l'intégration du système de traque de modifications, nous avons eu besoin de mettre à jour l'application. En effet, l'application fonctionnant sous Angular 15 durant l'ajout du système, nous avons eu besoin de la migrer à Angular 16. Avec la publication de l'application, nous lui attribuons différentes options afin d'optimiser son fonctionnement. Notamment, une fonction nommée Minification. Cette fonction réduit la taille du code source, de tout ce qui n'est pas utile pour son exécution. Cela comprend, les commentaires et espaces dans le code, la réduction de la taille des variables etc. Le problème étant que dans la traque de modification, nous nous basons sur le nom du composant. Ce nom étant lui-même réduit, deux composants peuvent désormais avoir le même nom. Ce qui crée des comportements non prédictibles, comme la sauvegarde du mauvais composant. Nous n'avons pour l'instant pas trouvé de solution en gardant le fonctionnement actuel.

Avant de sauvegarder des modifications, nous souhaitons formater les données. Par exemple, si l'utilisateur laisse une cellule vide dans une grille durant la sauvegarde, c'est un texte vide qui est envoyé. Cela peut créer des problèmes dans le fonctionnement de la logique métier du côté du serveur. En effet, le serveur attend une valeur nulle plutôt qu'un texte vide. Dans un autre exemple, si le serveur attend un Guid (Cf. glossaire) nullable et la grille contient un texte vide, nous voulons qu'une valeur nulle soit enregistrée. En revanche, une erreur est déclenchée lors de la sérialisation (Cf. glossaire) de la requête du côté du serveur car un texte vide ne correspond ni à une valeur nulle, ni à un Guid. C'est pourquoi j'ai mis en place un helper dans l'application web, permettant de formater dynamiquement des objets. Ce helper comprend une méthode prenant en argument un objet et des attributs. Ensuite il remplace les valeurs ciblées de l'objet par des valeurs nulles.

Dans une page nommée Layout, il a été nécessaire d'intégrer un sous système de navigation et de sauvegarde. La page se trouve dans un onglet comportant le système de sauvegarde décrit plus haut. Cet onglet est composé de trois grilles. La sélection d'éléments dans les grilles supérieures, définissent le contenu des grilles inférieures. Par exemple, si dans la première grille, nous sélectionnons un élément. Le contenu de la grille se trouvant en dessous change. De même pour la dernière grille tout en dessous. Le problème est que si les grilles inférieures comportent des modifications, après la sélection d'un autre élément dans une grille supérieure, elles peuvent être perdues. C'est pourquoi, un système de navigation a dû être intégré au sein d'une même page (reprenant le même fonctionnement que le système principal, avec l'affichage d'une modale de validation).

Et enfin, dans la page nommée Report Template, j'ai dû intégrer un sous système de navigation et de création. Tout comme dans la page de Layout, nous avons un onglet permettant de naviguer au travers de sous éléments. Nous devons également pouvoir créer de nouveaux sous éléments et en supprimer. Il a donc à nouveau été nécessaire d'intégrer un sous-système de sauvegarde.

5 Conclusion

En conclusion, le cahier des charges a été rempli et le système de sauvegarde est couramment utilisé par les employés de l'entreprise. De fait de sa souplesse permise par la refactorisation, le système de sauvegarde a été étendu aux nouvelles intégrations dans l'application et correspond aux besoins.

J'ai également appris durant la refactorisation, que le rôle du développeur n'est pas uniquement d'intégrer les fonctionnalités qu'on lui demande. Il doit également défendre l'intérêt pour la maintenance de l'architecture du projet. Pour le bien du projet, de son évolutivité, de sa stabilité et finalement de son coût.

Suite à la refactorisation du code et de l'architecture de l'application, l'intégration de nouvelles fonctionnalités est plus facile via la création et la réutilisation des composants de page, de formulaire et de grille. Et le fonctionnement de l'application est plus prédictible.

Néanmoins certains points peuvent encore être améliorés. Il aurait été utile de créer des abstractions de la grille pour la spécialiser en fonction des besoins. Car à de nombreux endroits nous définissons le même type de grille. De plus, une partie de code nécessite encore une refactorisation. Tout comme expliqué dans le chapitre 4.2.5 « Limitations rencontrées », la limitation de temps et de budget ont défini les frontières de la refactorisation.

Cette année d'alternance m'a également été bénéfique. En effet, j'ai eu l'occasion de travailler avec des développeurs qui m'ont partagé leur expérience. J'ai pu apprendre et mettre en œuvre les différents principes qui définissent la « Clean Architecture » vu par M. Robert C. Martin.

L'entreprise m'a proposé de renouveler pour la troisième année consécutive, mon contrat professionnel pour l'année 2023/2024.

L'apport de l'alternance à l'entreprise

« Complètement intégré à l'équipe Observo, Tom a su achever avec brio sa deuxième année d'alternance. Malgré les difficultés imposées par la langue, il a su travailler sans problème avec tous les intervenants de l'équipe en anglais et même en allemand lorsque cela a été nécessaire.

Poussant ses compétences Angular à un niveau supérieur, Tom a su mettre en place un système de sauvegarde robuste et évolutif dans une architecture complexe.

Le travail a été effectué dans les règles de l'art et suivant les standards les plus exigeants.

C'est avec sérénité et optimisme que nous accueillerons Tom pour sa dernière année d'alternance dans notre entreprise.» **M. Thibault Vellicus, Tuteur professionnel**

Résumé

Ce rapport résume mon travail accompli au cours de mon alternance sur le projet ObservoAdmin. Il commence par l'introduction qui comprend le choix de l'entreprise. Suivi de la description de mon parcours au sein de l'école UHA 4.0. Le tout suivit de la présentation de l'entreprise Unit Solutions et de ses différents projets, pour ensuite entrer dans le vif du sujet. Le sujet principal ici est la mise en place d'un système de sauvegarde dans une application web d'administration. Et enfin ce rapport se termine par la conclusion résumant l'utilité de cette alternance à l'entreprise et à moi-même.

Mots-clés

Refactorisation - Sauvegarde - Angular - Composant

Glossaire

Méthode d'Ostu : En vision par ordinateur et traitement d'image, la méthode d'Otsu est utilisée pour effectuer un seuillage automatique à partir de la forme de l'histogramme de l'image.

Histogramme (imagerie numérique) : L'histogramme d'une image est sa représentation graphique de la distribution de pixels.

Machine Learning : Le Machine Learning est un sous ensemble de l'intelligence artificielle. Il vise à apprendre aux machines à tirer des enseignements des données et à s'améliorer avec l'expérience.

Méthodologie Agile : La méthodologie Agile est une méthode basée sur des phases de développement de quelques semaines, pendant lesquelles le produit est développé.

Méthodologie Agile Scrum : La méthodologie Scrum est une démarche de gestion de projet qui fait du client (ou utilisateur) le principal pilote de l'équipe en charge des développements.

Daily meeting : La daily meeting est une courte réunion permettant à l'équipe Scrum de se synchroniser pour les 24 prochaines heures.

Epic : Un Epic est un regroupement de plusieurs User stories afin de représenter une demande trop grande pour être concentrée dans un ticket.

User story ou ticket : Une User Story ou un ticket est une explication générale d'une fonctionnalité logicielle écrite du point de vue de l'utilisateur.

Sprint : Un Sprint est une phase de développement dans la méthodologie Agile Scrum. Il contient une liste de tickets décrivant les tâches à effectuer durant le Sprint.

Smoke test (un test de fumée) : Un Smoke test consiste à tester une fonctionnalité d'une application. Au sein de Unit Solutions c'est un test effectué sur l'IHM afin de déceler des bogues.

Sprint retrospective : La Sprint retrospective est une réunion de l'équipe Scrum qui clôture un Sprint. Elle permet à l'équipe de proposer des axes d'améliorations pour les prochains Sprints.

Git : Git est un logiciel de gestion de version.

Compiler : Compiler est le procédé de traduction du code d'un programme lisible par l'humain en un code exécutable par une machine.

Test unitaire : Un test unitaire est un procédé visant à vérifier le bon fonctionnement d'une partie précise d'un programme.

Pull Request : Une Pull Request permet une vérification et un échange des développeurs au travers d'une interface web avant d'intégrer des changements dans un projet.

Client : Un client en informatique est le logiciel qui est utilisé par l'utilisateur. Par exemple, dans Observo, le client est l'application mobile.

Modèle : Un Modèle a pour objectif de structurer des informations.

Mandant : Un Mandant est un contrat qui définit pour qui les équipes utilisant l'application Observo travaillent. Par exemple certains clients sont des bureaux d'ingénieurs qui ont besoin de faire partie de plusieurs cantons ou plusieurs villes différentes. Quand un utilisateur fait partie d'un Mandant, cela lui permet d'avoir un accès aux ressources de travail lui correspondant.

Production : La production est l'environnement d'un logiciel utilisé par les utilisateurs.

Angular : Angular est un Framework permettant la création d'application web et mobile.

Asp .Net Core : Asp .Net Core est un Framework Web gratuit et Open Source développé par Microsoft et la communauté.

Open Source : Un logiciel Open Source est un logiciel ayant son code accessible à tous.

Backend : Un Backend est un service se trouvant sur le serveur qui est consommé par le client.

Framework : Un Framework est un ensemble de composants logiciels qui permettent de structurer et construire une application.

Hangfire : Hangfire est un projet Open-Source ayant comme objectif d'effectuer diverses tâches en arrière-plan.

BDD (Base De Données) : Une Base De Données permet de stocker et de retrouver des données via un programme.

IHM (Interface Homme-Machine) : Une IHM est une interface permettant à un utilisateur d'interagir avec un programme.

Service : Un Service dans Observo ou ObservoAdmin est un morceau de code proposant différentes fonctions réutilisables dans l'application.

Occultation : L'occultation d'une méthode dans la Programmation Orientée Objet signifie l'accessibilité de celle-ci. Elle peut-être accessible depuis l'extérieur (publique), inaccessible depuis l'extérieur (privée) ou accessible via l'héritage (protégée).

Classe (POO) : Une classe est un conteneur symbolique et autonome qui contient des informations et des mécanismes. C'est le concept central de la Programmation Orientée Objet.

POO : La Programmation Orientée Objet est un paradigme de programmation informatique. Elle consiste en la définition et l'interaction de briques logicielles appelés objets.

MVP (Minimum Viable Product) : La MVP ou Produit minimum viable en français correspond à la version d'un produit ou fonctionnalité permettant de remplir la demande avec un minimum d'effort.

Programmation par contrat : La programmation est un paradigme de programmation dans lequel le déroulement des traitements est régi par des règles.

Interface : Une interface dans le langage C# permet de mettre des données dans une forme prédéfinie.

String : Un String est une chaîne de charactères.

Boolean : Un Boolean est un type de valeur à deux états. Ces états sont « Vrai » ou « Faux ».

Cast : Un cast est un opérateur de conversion de type.

Logique métier : La logique métier fait référence au code concernant des fonctionnalités spécifique à des user stories.

Template ou vue : Le template d'une page correspond à son affichage.

Helper : Un Helper dans le projet Observo ou ObservoAdmin un morceau de code réutilisable permettant d'effectuer diverses actions propres au projet dans lequel il se trouve.

Méthode (POO) : Une méthode est une fonction se trouvant dans une classe.

Variable globale : Une variable globale est une variable partagée entre plusieurs fonctions souvent dans une même classe.

Débogage : Le débogage est un processus de diagnostic sur le fonctionnement d'une application.

Guard (Angular) : Les Guards dans Angular, permettent de limiter l'accès à une route.

Guid (identificateur global unique) : Un Guid est un entier de 128 bits qui est utilisé comme un identifiant. Sa taille lui permet d'avoir une probabilité minime d'être dupliqué.

Sérialisation : La sérialisation est le processus de conversion d'un objet en flux d'octets pour stocker l'objet ou le transmettre à la mémoire, une base de données ou un fichier. Son principal objectif est d'enregistrer l'état d'un objet afin de pouvoir le recréer si nécessaire. Le processus inverse est appelé désérialisation.

Inputs et Outputs (Angular) : Les Inputs et Outputs dans Angular permettent le transfert d'information entre les composants parent ou enfant.

Sources

https://unit.solutions/fr - 13/02/2022

https://unit.solutions/fr/notre-histoire - 13/02/2022

https://fr.wikipedia.org/wiki/M%C3%A9thode_d%27Otsu – 04/08/2023

https://fr.wikipedia.org/wiki/Histogramme_(imagerie_numérique) – 04/08/2023

https://www.hangfire.io/ - 05/06/2022

https://fr.wikipedia.org/wiki/Objet_(informatique) - 25/07/2022

https://www.atlassian.com/fr/agile/agile-at-scale/agile-iron-triangle - 11/07/2023

https://fr.abcdef.wiki/wiki/Project_management_triangle - 11/07/2023

https://monday.com/blog/fr/gestion-de-projet/le-triangle-de-gestion-de-projet-comment-ladapter-pour-mieux-planifier-vos-projets/ - 11/07/2023

https://fr.wikipedia.org/wiki/Principe_KISS - 21/07/2023

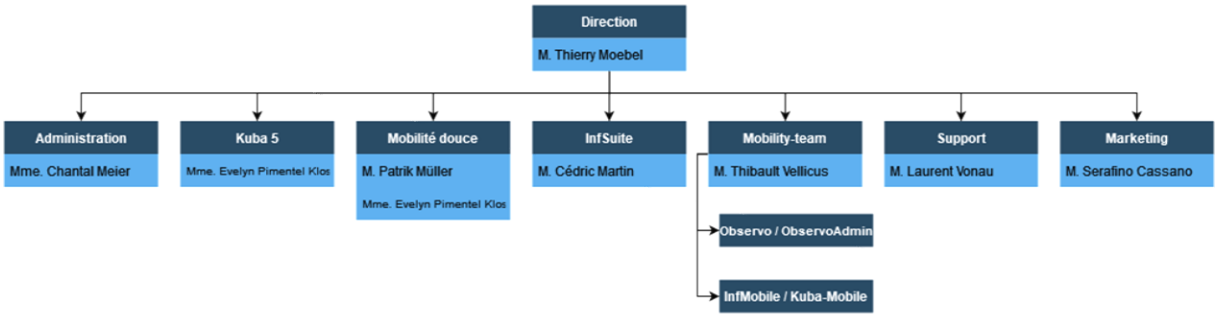
https://fr.wikipedia.org/wiki/Principe_de_substitution_de_Liskov#Conception_par_contrat – 26/07/2023

https://developer.mozilla.org/fr/docs/Glossary/Callback_function - 27/07/2023

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Promise - 27/07/2023

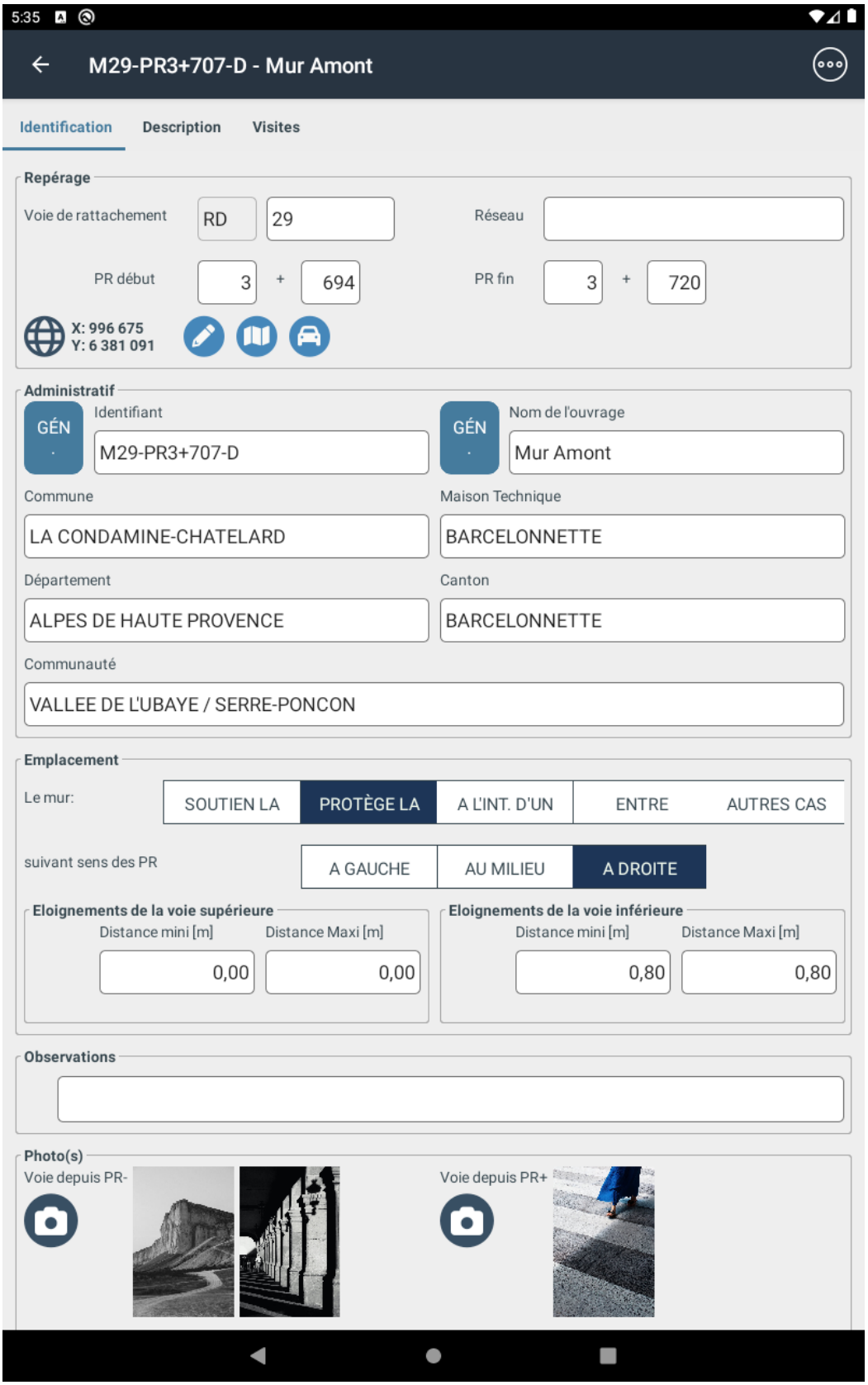
https://www.google.fr/books/edition/Clean_Architecture/uGE1DwAAQBAJ?hl=fr&gbpv=0&bsq=clean%20architecture 11/10/2022

Annexes



Annexe 1 : Organigramme représentant une vue globale des équipes dans l'entreprise Unit Solutions

Annexe 2 : Capture d'écran de la carte d'Observo avec des infrastructures inspectées.



Annexe 3 : Capture d'écran d'un formulaire dans Observo.

Implémenter un système de sauvegarde dans ObservoAdmin

Tom Willemin