République Algérienne Démocratique et Populaire
Université Abou Bakr Belkaid– Tlemcen
Faculté des Technologies
Département de génie éléctrique

Mémoire de fin d'études

pour l'obtention du diplôme de Master en Génie Industriel

Option: *Ingénieur des systèmes*

Thème

Architecture des systèmes d'information avec étude de cas : **Interface de contrôle d'accès sécurisé pour les établissements.**

Réalisé par :

- **Tasfaout Yassine**

Présenté le 27 Juin 2018 devant le jury composé de.

Jury	Grade	Position	Établissement
BETAOUAF Hichem	**Président**	MCB	Université de Tlemcen
Mme BENSMAINE Nardjes	**Examinatrice**	MCB	Université de Tlemcen
BESSENOUCI Hakim Nadhir	**Examinateur**	MAA	Université de Tlemcen
BENSMAIN Yassir	**Invité**	Doctorant	Université de Tlemcen
MALIKI Fouad	**Encadrant**	MAA	ESSA-Tlemcen
BELKAID Fayçal	**Encadrant**	MCA	Université de Tlemcen
BAGHLI Fayçal	**Co-Encadrant**	Directeur	NetSprint

Remerciements

Tout d'abord, je remercie Allah, de m'avoir donné la Foi et le courage d'accomplir ce modeste travail. Je voudrais remercier mes parents de me demander sans relâche si j'ai terminé mon mémoire. Je ne crois pas au concept qu'un homme ne doit sa réussite qu'à lui-même. Je suis arrivé à ce stade de ma vie grâce à l'aide de nombreuses personnes qui ont voulu me voir réussir ; mes enseignants, mes collègues et aussi les membres qui veillent sur la réussite de la filière ***Génie Industrielle****. Toutefois, je tiens à remercier spécifiquement mes encadrants* ***Dr MALIKI Fouad****,* ***Dr BELKAID Fayçal*** *d'avoir accepté de m'encadrer c'était un honneur de travailler avec eux, je les remercie pour m'avoir fait confiance et pour leur patience avec moi. Et Je tiens à remercier* ***M. BAGHLI Fayçal*** *mon superviseur à* ***NetSprint*** *de m'avoir soutenu et donné la flexibilité de gérer ce projet avec l'aide de* ***Mlle BAGHLI Zineb*** *qui suivait l'avancement du projet sur* ***Kanban****. Et enfin, Je tiens à remercier les membres du jury et à montrer mon profond respect et ma gratitude pour eux d'avoir accepté d'évaluer cette recherche, et de permettre cette graduation.*

Table des matières

Introduction Générale

Les systèmes d'information sont devenus de plus en plus complexes. Des millions de lignes de code, des Petabytes de données représentées dans des milliers de tableaux reliés entre eux ou des documents, et plusieurs composants ou services, tous exécutés sur plusieurs serveurs éparpillés un peu partout dans le monde. Décidément, les structures des systèmes d'information sont parmi les structures les plus complexes que l'être humain a construit. Cela est arrivé à cause du besoin exponentiel des entreprises de garder, transférer et analyser de larges portions de données car leur processus et leur service en dépend fortement. Par conséquence, des défis complexes émergent pour les développeur et ingénieurs pour maintenir et faire évoluer ces systèmes. De là provient le besoin de l'architecte ou du groupe d'architectes des systèmes dans une organisation, où ils gèrent l'intégralité du système avec une vue macroscopique sur ce dernier. Le rôle de l'architecte des systèmes dans une organisation peut prendre plusieurs formes, mais fondamentalement, nous pouvons dire qu'il est là pour guider l'organisation en matière de **IT**[1] et trouver des solutions aux défis auxquels l'organisation fait face.

L'objectif primaire de ce mémoire est de créer un système de base qui a pour objectif d'être extensible. Celui-ci à son tour s'étendra en plusieurs systèmes de gestion d'entreprise comme dans les domaines de l'hôtellerie, la restauration, le commerce ou enfin le système traité dans ce cas d'étude : l'interface de gestion d'accès des serrures RFID [2]connectés.

Ce travail est étalé sur une feuille de route, qui montre notre parcours dans ce mémoire pour arriver à notre système final. Chaque chapitre traite une partie du cycle de vie d'un système d'information, en commençant par la définition du besoin que le système doit à satisfaire, les modèles d'architecture et leur procédure, les méthodes de travail et de développement dans l'industrie du logiciel, puis plus de détails avec les concepts de programmation des logiciels évolutifs, et enfin notre cas d'étude et comment nous avons implémenté les connaissances qui précèdent.

[1] Information Technology
[2] Radio Frequency Identification

Chapitre 1 : Modèles d'architecture

Ce chapitre donne plusieurs définitions sur l'architecture des systèmes d'information. Puis argumente le besoin crucial d'avoir une architecture quelle que soit la taille du système. Et il détaille les modèles d'architecture utilisés par l'industrie tel que ADD (Attribute Driven Design).

Chapitre 2 : Les concepts d'architecture

Ce chapitre aborde les concepts d'architecture tel que Client-Server, Layer Pattern ou bien Pipe-Filter....

Chapitre 3 : Les modèles de développement dans les projets des systèmes d'information

Ce chapitre met en évidence les différentes méthodes de travail et de développement en Waterfall ou Agile, en présentant les méthodes KANBAN et Scrum. De plus nous discuterons les bonne pratique de développement et le principe **SOLID**.

Chapitre 4 : Étude de cas – Implémentation du ADD

Ce chapitre représente l'implémentation du système finale, du modèle d'architecture, outils et technologies utilisé au résultat final et les perspectives d'évolution du système.

État de l’art

1. Introduction

Ce chapitre représente un aperçu sur la littérature dans le domaine de l'architecture des systèmes d'information, modèles de travail et l'impact des bonnes pratique du développement et son impact sur la qualité des systèmes. Cette partie du mémoire cite les différents publications sur les aspects mentionné précédemment.

2. Méthodes de conception d'architecture

2.1. ADD Method

Attribute Driven Design développé par la collaboration entre Carnegie Mellon University et Robert Boosch, GmBH. C'est un modèle généralisé pour une architecture de haut niveau des systèmes. Elle se focalise sur les étapes initiales de conception des systèmes où elle facilite le départ du processus même avec le manque de détails des exigences. Elle est conçue pour être adéquate avec une grande variété de domaines. Enfin sa nature itérative la rend très attractive pour les équipes agiles. [1]–[3]

- **Objectif** : ADD est une méthode pour définir l'architecture des systèmes d'information qui se base sur les attributs de qualité. Avec un processus récursif elle permet de passer par plusieurs itérations où chaque itération présente un objectif et des attributs à satisfaire. Son objectif est d'arriver à une architecture initiale rapidement tout en satisfaisant les attributs de qualité définis.

- **Procédure** :

La méthode ADD suit le modèle Plan-Do-Check [3]

- **Plan** : On prend les attributs de qualité et les contraintes et quels éléments architecturaux qui peuvent les satisfaire
- **Do** : Les éléments sont instanciés pour satisfaire les attributs fonctionnels et non-fonctionnels choisis.
- **Check** : Le résultat est analysé pour déterminer si les exigences sont résolues.

2.2.Rational's Architecture Design Method

Développé par Rational® appelé Rational Unified Process qui est un framework qui permet de développer et de déployer les systèmes d'information et l'organisation des équipes de développement.[4]

- **Objectif** : Avoir un modèle qui couvre tout le spectre développement ; le cycle de vie du système, méthode de travail et organisation d'équipe. Avec les bonnes pratiques suivantes :
  - Développer d'une façon itérative.
  - Gestion des exigences et leur changement.
  - Utiliser l'architecture des composants.
  - Modéliser l'architecture visuellement UML [3](Originalement créée par Rational Software, à présent maintenu par Object Management Group OMG).
  - Vérifier périodiquement la qualité du système.
  - Contrôler les changements du système.
- **Procédure**

L'analyse RUP est conçue autour ces trois entités :

- **Workers | Chargé** « Qui ? »
- **Artifacts | Artefact** « Quoi ? »
- **Activities | Activités** « Comment ? »

L'architecte communique les activités techniques et construit le squelette de l'architecture et les différentes vues du système. Puis ces artefacts seront des entrées pour les développeurs pour affecter les responsabilités aux éléments identifiés. Tous les artefacts sont capturés dans le document d'architecture.

[3] Unified Modeling Language

2.3. Goal-Directed Design Method [5]

La création de cette méthode a vu le jour après avoir réalisé que le souci primaire de chaque système est d'accomplir son objectif. Cette méthode se focalise sur la décomposition de l'objectif primaire du système en plusieurs sous-objectifs intermédiaires.

- **Objectif**

Cette méthode a trois objectifs spécifiques :

  - Supporter une architecture de haut niveau.
  - Faciliter la réutilisation des éléments software qui atteignent les mêmes sous-objectifs.
  - Unifier les résultats entre les développeurs et concepteurs.

- **Procédure**

Le processus de cette méthode est établi sur quatre étapes itératives :

  - Identifier l'objectif clé du système.
  - Identifier les abstractions de haut niveau qui peuvent satisfaire l'objectif clé du système.
  - Identifier les sorties des sous-objectifs qui à leur tour seront des entrées pour l'objectif clé du système.
  - Pour les entrées de l'objectif clé, définir s'il y a des contraintes sur le flux de données.

2.4. Discussion

Pour récapituler ces méthodes servent des objectifs commun tel que :

- Satisfaire les attributs de qualité.
- Capturer et formaliser l'architecture.
- Fournir plusieurs vues sur le système.
- Séparation du flux de données du comportement du système.
- Le système doit être basé sur un objectif.

ADD prend en charge les attributs de qualité tout au long de son processus, pour éviter de les laisser en fin des itération qui est les cas de plusieurs d'autre méthodes. Pour tous les méthodes citées ils prennent tous en charge une façon de gérer les vues de l'architecture ; cas d'utilisation, architecture de références et aussi **UML** tout au long de leur procédure pour deux objectif ; documenter l'architecture et arriver à l'architecture finale à partir l'évolution de ces vues. **RUP** est la plus complète dans le sens où elle gère tous les aspects de l'architecture plus l'organisation d'équipe et offre des outils et des lignes directrices pour chacun et chaque activité. **Goal-Directed Design Method** essai de résoudre un problème récurrent dans le développement des systèmes d'information, qui est la divergence du système de son objectif en essayant de prédire des exigences futur.

3. Modèles de travail

3.1. Kanban et son utilisation de l'industrie d'informatique

À l'ère du changement constant dans le domaine du business, les organisations se trouvent dans la course d'adaptation continue pour répondre à ces nouvelles demandes, tel que la nécessité de réduire le temps d'arrivé au marché et développer des produits avec meilleur qualité en réduisant les couts. De nombreuse approches on étaient établit pour instituer des solutions aidant les organisations à répondre à ces changements. Spécifiquement, à l'industrie informatique les méthodes agiles ont été suggérés comme solution, ces derniers ont prouver d'être efficace pour les projet complexes avec un haut degré d'imprévisibilité, où les méthodes Plan-driven n'étaient pas à la hauteur. Les méthodes agiles tel que Kanban sont dérivées depuis l'approche Lean Software Development à son tour originaire depuis le Lean Manufacturing avec l'objectif d'éliminer tout type de déchets. Dans le domaine de création des système d'information être Lean s'agit d'éliminer tous qui n'ajoute pas de valeur aux parties prenantes. L'utilisation du Kanban dans le développement des systèmes d'information est de plus en plus populaire chez les praticiens du développement des systèmes informatique , pour de nombreuses raisons identifiées par la revue de littérature[6] ; une amélioration sur le temps de cycle de la réalisation des projets, augmentation de qualité dans les produits finaux, meilleur coordination et communication entre les équipes, et enfin une diminution des défauts signalés par les clients.

Principes du Lean Software Development	Principes du Kanban
Éliminer les déchets	Visualiser le flux de travail
Qualité	Limiter le **Work In Progress**
Créer de la connaissance technique	Mesure et benchmark du flux
Livré rapidement	Améliorer la collaboration
Respect des autres	
Optimisation en tout	

*Tableau 1 État de l'art - **Lean et Kanban*** [6]

L'application du **Kanban** devrait être positive due aux résultats positifs du **Lean Manufacturing**, cette espérance est aussi basée sur les caractéristiques du Kanban à être flexible, visuelle et favorise la communication et collaboration au sein de l'équipe. Le tableau suivant cite les références des études, articles de conférence qui supportent les avantages du Kanban.

Avantage du KANBAN	Articles
Meilleur compréhension du processus de développement	[7]–[12]
Meilleur qualité du produit finale	[7], [11], [13]–[16]
Motivation des ingénieurs	[7], [9], [10], [16]–[18]
Le concept du WIP aide à libérer la chaine du processus	[7], [9], [10], [12], [19]
Augmentation de la productivité	[7], [10], [16], [20]
Livraison rapide	[7], [20]
Documentation minimale	[9]
Meilleur Communication	[9]–[11], [14], [21]

*Tableau 2 État de l'art - **Les avantages de Kanban*** [6]

3.2. Discussion

Plusieurs articles ont été publiés sur Kanban dans les dernières années,ce qui est un indicateur de tendance croissante dans l'industrie informatique. Toutefois, l'impact du Kanban ne peut pas être évalué correctement pour le moment car il y a un manque de recherche qui qualifie Kanban d'une façon qualitative et quantitative. De plus, sa flexibilité doit être accompagné avec des lignes directrices pour permettre aux organisations de savoir qu'elle est la meilleur implémentation du Kanban dans leur propre contexte.

4. Les modèles de développement

4.1. Les bonnes pratiques du développement et leur impact sur la qualité des systèmes informatique

L'une des tâches les plus importantes du développement des systèmes informatique est d'assurer la qualité du code qui constitue le système. La qualité d'un système est déterminé par les caractéristiques non-fonctionnelles du système. Pour cela, des modèles sont établis pour déterminer ces caractéristiques et leurs métriques. Un des modèles établis est **ISO/IEC 9126-1**[22] qui définit les caractéristiques de qualité du software.

Caractéristiques	Sous-Caractéristiques
Fonctionnalité	Pertinence, Précision, Interopérabilité, Sécurité
Fiabilité	Maturité, Tolérance aux pannes, Récupération
Utilisabilité	Compréhensibilité, Apprentissage, Opérabilité, Attraction
Efficacité	Comportement temporel, Utilisation des ressources
Maintenabilité	Analysabilité, Changeabilité, Stabilité, Testabilité
Portabilité	Adaptabilité, Installation, Coexistence, Remplaçabilité

*Tableau 3 État de l'art - **ISO 9126-1 Modèle de caractéristique de qualité*** [22]

4.2. Maintenabilité

La maintenabilité permet de réduire les coûts et les efforts de changement d'un système lors d'un changement d'exigence. Toutefois, il n'est pas facile de développer un système hautement maintenable. Dans la littérature, on trouve de nombreuses approches qui présentent des modèles et des design patterns[4] et leur impact sur cet attribut de qualité. **Prechelt et al**. [23] décrivent une expérience où ils ont adressé plusieurs **design patterns** tels que Decorator, Composite, Abstract Factory, Observer, et Visitor. Ils ont trouvé qu'il est toujours préférable d'utiliser des patterns même s'il y a une solution plus simple car ces patterns ont tendance à donner plus de flexibilité pour les décisions imprévisibles.

[4] Modèle de développement pour résoudre un problème récurrent.

Le tableau suivant récapitule une portion des études qui ont porté sur les designs patterns et leur impact sur la maintenabilité [24]

Type de Pattern	Attribut de Qualité	Maintenabilité						
	Pattern	[22]	[25]	[26]	[27]	[28]	[29]	[30]
Créationnel	Abs. Factory	X	X	X	X		X	
	Singleton							
	Builder							
Structurel	Adapter							
	Composite	X	X	X	X	X	X	X
	Decorator	X	X	X	X	X	X	
Comportemental	Chain of responsibility							X
	Observer	X	X			X		
	Visitor	X	X			X		

Tableau 4 État de l'art - ***Une portion d'un tableau sur le reportage des études sur les design patterns et leur impact sur les attributs de qualité***[24]

4.3. Discussion

L'utilisation des design patterns a prouvé son efficacité à améliorer les attributs de qualité du programme qui constitue le système. Cependant, le développeur doit faire attention à ne pas utiliser la mauvaise pattern dans la mauvaise situation ; en fonction du contexte du problème il y a un ou quelques patterns qui le résolvent en favorisant certains attributs de qualité. Enfin, l'un des avantages des pattern c'est qu'ils sont populaires et leur utilisation permet à un nouveau développeur dans le projet de comprendre facilement la **Codebase**[5].

[5] Code Source d'un système

Chapitre I

MODÈLE D'ARCHITECTURE ET DE DÉVELOPPEMENT.

1. Introduction

L'architecture logicielle est parmi les termes qui sont utilisés obscurément dans le domaine de l'IT. Plusieurs définitions de l'architecture software ont été publiées dans la littérature, telle que la définition de l'IEEE.

> « *L'architecture est définie par la pratique recommandée comme l'organisation fondamentale d'un système, incorporé dans ses composants, leurs relations les uns avec les autres et l'environnement, et les principes régissant sa conception et son évolution*. » [31]

De cette définition on comprend que la discipline de l'architecture de software capture la structure du système en détaillant ses composants et leurs interactions. Elle définit aussi les règles auxquelles le système doit adhérer, et finalement l'évolution du système dans le temps.

Une autre définition émise par le livre référence sur l'architecture des logiciels **Software Architecture In Practice :**

> « *L'architecture d'un programme ou d'un système informatique est la structure ou les structures du système, qui comprennent des éléments logiques, les propriétés visibles à l'extérieur de ces éléments et les relations entre eux.* » [31]

Cette définition rajoute le point d'abstraction et comment l'architecture dois viser l'abstraction et définir à quel point les éléments seront visibles à l'environnement extérieur. Cette définition prouve la transition que le monde du software design a traversé ; du développement de bibliothèques à l'élaboration des API[6] avec plus d'abstraction et de facilité d'utilisation.

Dans ce chapitre on discutera du processus de l'architecture des logiciels, en résumant le rôle d'un architecte de software et les activités du software design.

[6] Application Programming Interface

2. Rôle d'un architecte de système

Dans le domaine de l'ingénierie du software l'architecte joue un rôle crucial, car il est responsable d'initier, de suivre et d'achever le processus de conception. On verra comment ces activités de conception sont établies dans les sections qui suivent. De plus, l'architecte doit travailler avec son équipe sur l'étude des exigences, où ils doivent faire la collecte des exigences fonctionnelles et non-fonctionnelles et éliciter ces dernières. L'architecte doit aussi travailler étroitement avec les parties prenantes pour assurer que leurs besoins de l'application sont bien satisfaits. En outre, l'architecte doit aussi guider l'équipe de développement et de conception en définissant l'architecture du système et son plan de réalisation. Ce qui nous mène à la dernière obligation d'un architecte ; collaborer avec les gestionnaires du projet, aider à la planification, l'estimation et l'allocation des tâches.

3. Résumé du processus [31]

Le rôle de l'architecte est fastidieux. Afin de guider les architectes à travers leur processus de conception, il y a des modèles à suivre, généralement définis dans un processus itératif en trois étapes ; Définition des exigences, Elaboration d'architecture et terminer avec une validation.

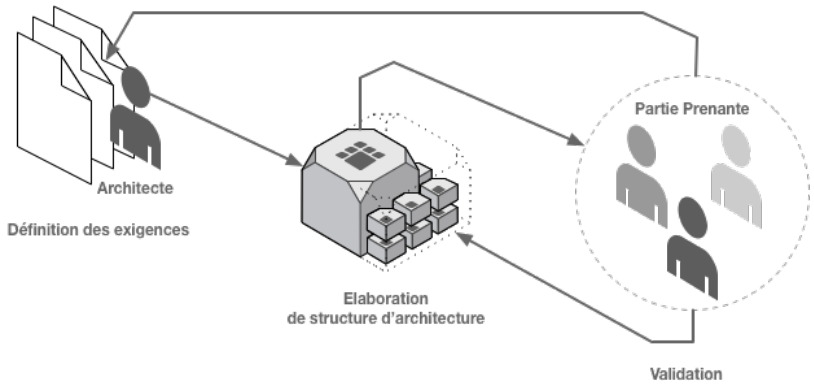

Figure 1 ***Processus générale de la conception d'un système***

La nature itérative de ce processus permet à l'architecte de valider sa conception dans le besoin d'une modification ou bien une nouvelle exigence qui doit être implémentée. Ceci le rend pratique pour les modèles de développement Agile ou Waterfall.

3.1. Définitions des exigences

Pour élaborer une architecture d'un logiciel, il faut tout d'abord avoir les exigences fonctionnelles et non-fonctionnelles bien établies, et bien sûr comprises. Ces exigences-là peuvent avoir plusieurs sources et formes, dans cette étape l'architecte doit rédiger un document qui va cataloguer toutes ces exigences et pour les identifier, cela nécessite de la part de l'architecte qu'il parle avec toutes les parties prenantes pour identifier les besoins explicites et aider à démystifier les exigences implicites ou ambiguës.

Attributs de Qualités	Exigences
Performance	97% des requêtes doivent être traitées avec succès.
Sécurité	Chaque action d'utilisateur doit être authentifiée.
Convivialité	L'interface doit être accessible depuis le web et mobile
Scalabilité	Plus de 500 utilisateurs connectés en même temps

Tableau 1 Résumé du processus - ***Exemple des exigences***

3.2. Priorisation des exigences[31]

Après avoir établi les exigences, il est nécessaire de les prioriser. Pour atteindre cela il y a plusieurs méthodes proposées par des modèles approuvés tel que ADD[7] ou DDD[8]. Cependant, on va voir comment prioriser en général avant d'entamer les modèles cités. Généralement, les priorités des fonctionnalités sont catégorisées en :

- **High** : Les fonctionnalités à supporter impérativement, ces fonctionnalités vont piloter l'architecture
- **Medium** : Ces fonctionnalités doivent être satisfaites à un certain moment dans un des livrables, pas nécessairement dans le premier livrable.

[7] Attribute Driven Design
[8] Domain Driven Design.

- **Low** : Ces fonctionnalités sont les commodités, leur ajout est un plus désirable. Mais ils ne sont pas les pilotes de l'architecture.

En priorisant il y aura des conflits entre des fonctionnalités telles que « Disponibilité du système et utilisation d'un nombre limité de serveurs. » « Arriver en premier sur le marché et stabiliser ensuite. » Ces conflits font la difficulté du rôle d'architecte, et pour les résoudre, l'architecte doit arriver à des compromis « Trade-off » avec les parties prenantes.

3.3. Élaboration de structure architecturale

Il est évident que parmi toutes les tâches de l'architecte, la plus pénible c'est de trouver l'architecture qui répond à tous les exigences. Réellement, une architecture répond aux exigences primaires d'une façon complète et partiellement aux autres. Pour aboutir à une architecture on doit choisir entre plusieurs concepts de design approuvés et testés auparavant pour satisfaire des exigences données. Généralement, pour une petite ou moyenne application, un seul concept suffira, mais pour un système complexe, il faudra plusieurs concepts assemblés ou hybridés, pour achever les besoins du système en question. Le choix du bon concept à implémenter nécessite une expérience mais aussi une connaissance des détails de chaque concept, leurs atouts et compromis. On verra quelques design concepts dans les sections qui suivent.

3.4. **Validation**[31]

La validation d'une architecture pose des complications nombreuses, car quel que soit l'architecture pour un nouveau système ou bien une évolution d'un système existant, une conception en elle-même ne peut être testée que si elle est implémentée et exécutée. Il est aussi fortement possible qu'un système intègre des sous-systèmes non-testables, boîte noire qui rend le système très difficile à tester. Pour remédier à la non-testabilité d'une conception, il y a deux techniques utilisées qui ont prouvé leur efficacité ; **Scénarios de Test** et **Prototypage.**

3.4.1. Scénarios de test

Cette technique consiste à poser des questionnements envers les attributs de qualité de l'architecture, elle suppose le pire des cas pour voir qu'elle est la réponse de l'architecture et comment elle se comporte dans de telles circonstances.

Attributs de qualités	Scénarios	Réponse
Disponibilité	Serveur d'email échoue	Les messages sont gardés dans un système de file d'attente, d'où ils seront envoyés après le redémarrage du serveur.
Sécurité	Aucune requête reçue d'un utilisateur dans les dernières 10 min	Le système traite cette session comme une session expirée et déconnecte l'utilisateur.
Scalabilité	Charge des requêtes se double depuis la période d'inscription	Le load-balancer commence à dispatcher les requêtes vers plusieurs serveurs pour équilibrer la charge.

Tableau 2 Résumé du processus - ***Exemple des scenarios de test***

3.4.2. Prototypage

La technique des scénarios est efficace, mais malheureusement il y a des tests qui ne peuvent être confirmés que par un système concret. D'où vient l'utilité du prototype. Les prototypes sont des versions minimales du système en question. On peut distinguer deux types de prototypes ; un prototype qui prouve un concept ou bien un prototype qui prouve une technologie.

Le prototype est un très bon outil de test et donne des résultats concrets pour valider une architecture d'un système.

4. Modèle d'architecture ADD

On discutera de manière détaillée la méthode de conception des systèmes d'information ADD[9]. On débutera avec un aperçu général sur ses procédures, puis on abordera les différents aspects à considérer dans chaque étape. De plus on parlera des différentes techniques d'identification et de sélection des concepts du system design, de l'implémentation de ces concepts, de la mise en œuvre de la documentation et enfin comment suivre l'avancement du processus de conception.[1]

4.1 Aperçu

L'architecture des systèmes d'information est généralement étalée sur des phases, des itérations aussi appelées incréments. Dans chacun de ces incréments, des activités de conception sont achevées. La méthode ADD présente l'avantage d'avoir les détails exhaustifs pour chacune de ces étapes. La particularité du ADD par rapport aux autres méthodes est sa concentration sur les attributs de qualité et leur aboutissement, et aussi son objectif de combiner l'analyse avec la documentation. ADD existe depuis 2001 et a connu autant d'améliorations, menant à la publication du ADD 2.0 en 2006, puis l'apparition du ADD 3.0 qui est la version actuelle du modèle ADD représenté dans la *figure 3***.** Avant d'entamer les étapes du ADD on doit avoir une fondation sur comment définir les pilotes car ils sont les entrées de notre flux de conception. Avoir les entrées correctement définies permettra à notre processus de générer le résultat attendu.

4.2 Pilotes (Entrées)

4.2.1 Objectif de conception

L'objectif d'architecture doit être clairement communiqué, pour quelle raison on veut réaliser cette architecture, et quel objectif d'affaire on veut satisfaire. En définissant l'objectif de conception, nous pouvons être dans ces trois situations ; proposition de projet,

[9]Attribute Driven Design

création d'un prototype exploratoire, ou d'une architecture en cours de développement, soit un nouveau système ou une partie d'un système existant qu'on veut améliorer.

1) Proposition de projet : L'objectif est d'avoir une architecture avec suffisamment de détails pour aider à l'estimation de la faisabilité, du planning et du budget.
2) Prototypage : L'objectif n'est pas d'avoir une architecture d'un système livrable ou réutilisable, mais une architecture qui aide à explorer le domaine d'exploitation, ou pour examiner des attributs non-fonctionnels tels que performance, évolutivité ou disponibilité du système.
3) Au cours du développement : Lors de la création d'un nouveau système ou l'amélioration d'un système existant, l'objectif est d'achever suffisamment de design pour satisfaire les exigences, et assister la production du système.

Selon le type de système et la maturité de son domaine, l'objectif de conception est interprété différemment. Système Greenfield dans un domaine mature, système Greenfield dans un nouveau domaine, ou système existant Brownfield. Supposons que vous êtes dans un domaine mature, l'architecture est relativement simple ; on peut utiliser des systèmes existants et faire des estimations. Si vous étiez dans un nouveau domaine, le processus est beaucoup plus risqué, dans de telles circonstances, un prototype du système est la clé pour réduire l'incertitude et explorer le domaine. Dans le système Brownfield, bien que les exigences soient mieux comprises au préalable, le système lui-même peut être complexe et doit être compris pour parvenir à un planning adéquat. De plus, les objectifs de l'organisation peuvent également affecter l'objectif de conception. Comme votre organisation peut être intéressée par la conception pour la réutilisation, la conception pour les extensions futures ou la conception pour la livraison continue. L'organisation joue un rôle clé dans les décisions d'architecture. Par conséquence, l'architecte doit établir un objectif de conception clairement communiqué, car il peut affecter l'ensemble du processus de conception.

4.2.2 Attributs de qualité (QA)

Def : *Les attributs de qualité sont des propriétés non-fonctionnelles mesurables et testables. Ils sont utilisés pour indiquer dans quelle mesure le système répond aux exigences des parties prenantes.*[1]

Étant donné que la qualité a tendance à être subjective, ces attributs sont présents pour l'exprimer de manière objective. Parmi tous les pilotes de conception, les attributs de qualité sont ceux qui affectent le plus l'architecture de manière considérable. Les décisions de conception critiques que vous allez prendre détermineront si votre architecture répondra ou non à ces attributs. Par conséquence, vous devez soigneusement éliciter, spécifier, prioriser et valider ces attributs de qualité. Cela rend la définition des attributs de qualité une tâche délicate. Heureusement, il existe des outils éprouvés et largement utilisés pour vous aider.

Quality Attribute Workshop (QAW) est une réunion de brainstorming entre les architectes et les parties prenantes, pour aboutir à la définition des attributs de qualité.

Utility Tree (Arbre Utilitaire) outil pour prioriser les attributs de qualité.

Selon la littérature de l'architecture des systèmes, la meilleure façon de décrire un attribut de qualité est de le formuler dans un scénario compréhensible. Avec le modèle suivant :

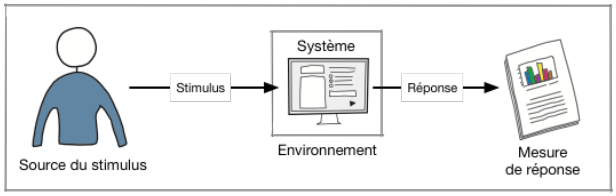

Figure 2 ***Scénario de qualité***

Exemple :

Prenons l'exigence fonctionnelle suivante et traduisons-la en attribut de qualité. Admettons que nous sommes en train de développer un système bancaire et notre exigence fonctionnelle est « Le client peut faire une transaction depuis son compte à un autre compte. « En attribut de qualité » Le client peut faire une transaction de compte à un autre compte en « 5min » Où la source du stimulus est le client, le stimulus est la demande de transaction, le système est le système de transaction et l'environnement est le mode du

système ; mode optimal, dégradé ou d'autres modes. Enfin, la réponse, qui est le succès ou échec de la transaction.

Les attributs de qualité sont mesurables sur deux dimensions, pour chaque dimension on assigne un dégré d'importance :

- Dimension Client (**[L]**ow/**[M]**eduim/**[H]**igh)
- Dimension Risque Technique **([L]**ow/**[M]**eduim/**[H]**igh)

Exemple : QA-1 (H, H) ; QA-2(H, M)

Après avoir donné le classement aux attributs de qualité, ils doivent être priorisés de la plus haute importance (H, H), (M, H) ou (H, M). On peut utiliser un « **Utility Tree »** pour organiser les attributs de qualité.

Exemple :

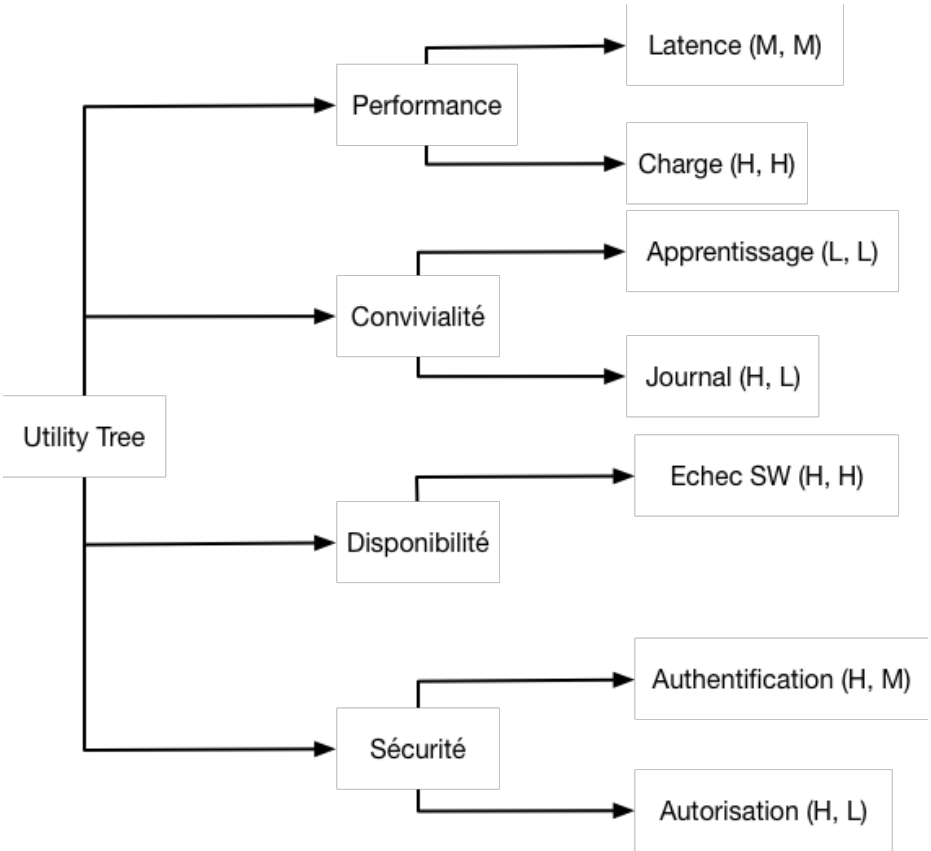

Figure 3 ***Arbre utilitaire*** [1]

4.2.3 Fonctionnalité Primaire

Il est impératif de considérer la fonctionnalité primaire du système lors de l'architecture, car elle est critique pour aboutir à l'objectif d'affaire du système.

Généralement, la fonctionnalité primaire est celle avec le plus haut degré de difficulté technique, ou celle qui demande plusieurs interactions entre les éléments architecturaux. En règle générale, 10% des cas d'utilisation sont des fonctionnalités primaires.[1]

Identifier la fonctionnalité primaire aide à allouer cette fonctionnalité aux éléments de l'architecture du système. Aussi, dans la plupart du temps cette fonctionnalité est directement liée à un ou plusieurs attributs de qualités, ce qui fait que lors de la prise de décision pour supporter l'attribut de qualité, l'architecte doit s'assurer que la décision prise n'affecte pas la fonctionnalité liée.

4.2.4 Soucis Architecturaux

Ce sont les aspects supplémentaires de l'architecture qui ne sont pas exprimés dans les exigences. Tels que la structure des modules, l'allocation des fonctionnalités aux modules, la gestion interne des dépendances, l'authentification, l'autorisation, le déploiement, la maintenance et d'autres.

4.2.5 Contraintes

Ce sont les déviations que vous devez prendre. Ils peuvent prendre une forme de vieilles technologies que vous devez utiliser, un système complexe avec lequel votre nouveau système doit communiquer, la disponibilité des développeurs, les délais, la rétrocompatibilité ou le respect de certaines normes. Tous ces éléments doivent être catalogués et pris en compte pendant le processus d'architecture.

4.3 Étapes du modèle ADD

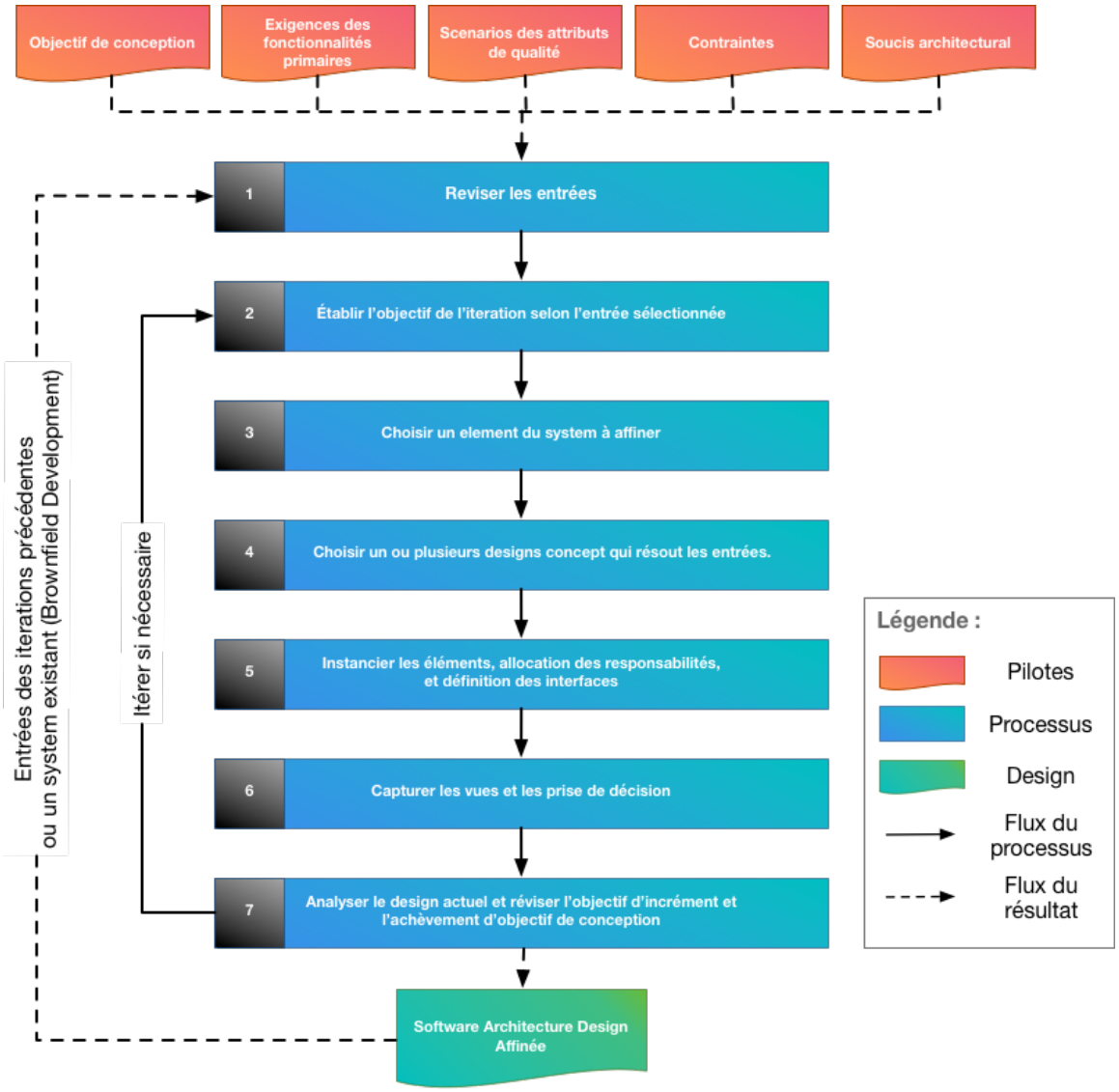

Figure 4 ***Processus ADD 3.0***

4.3.1 Étape 1 « Réviser les entrées »

Tout d'abord, assurez-vous que toutes vos entrées sont disponibles. À présent, vous devez être clair sur votre objectif de conception, si vous concevez pour une estimation précoce, affinez un design existant ou générez un prototype pour atténuer certains risques techniques. Ainsi que les autres pilotes tels que fonctionnalité primaire, attribut de qualités

et contraintes. Dans cette étape, vous devez analyser tous les pilotes collectés et voir si vous avez raté un détail en cours de route, comme une partie prenante du projet qui a changé ses exigences ou si vos priorités sont bonnes. Après cela vous pouvez commencer une itération de design.

4.3.2 Étape 2 « Établir l'objectif de l'itération selon le pilote sélectionné »

Typiquement à chaque phase de conception, nous effectuons une série d'itérations de conception et chaque itération vise un objectif particulier, et est destinée à satisfaire un sous-ensemble de pilotes. Par conséquent, en fonction de l'itération où nous sommes, nous aurons un objectif spécifique à atteindre. Par exemple, si on est dans la première itération dans un système Greenfield dans un domaine mature, l'objectif d'itération sera d'avoir une structure globale.

4.3.3 Étape 3 « Choisir un élément du système à affiner »

Pour satisfaire les pilotes, vous devez produire plusieurs structures architecturales, qui sont composées d'éléments interconnectés. Ces éléments sont obtenus grâce à la décomposition d'autres éléments identifiés dans une itération antérieure. Pour le système Greenfield, vous pouvez commencer par le système entier lui-même comme premier élément à décomposer. Dans les systèmes existants ou l'itération de conception ultérieure, vous choisissez des éléments déjà identifiés.

4.3.4 Étape 4 « Choisir un ou plusieurs concepts de design pour résoudre le pilote sélectionné »

Le choix d'un design concept est de loin la tâche la plus décourageante dans l'architecture du système, car vous aurez à faire face à de nombreuses alternatives. On verra dans la section des design concepts les différents types et leurs multiples options.

4.3.5 Étape 5 « Instancier les éléments architecturaux, allocation des responsabilités et définition des interfaces »

Après avoir fait un choix sur le design concept à implémenter pour satisfaire un certain pilote (fonctionnalité, attribut de qualité, contraintes ou soucis architectural), l'étape suivante sera d'instancier les éléments de ce design concept. Selon votre choix il y aura des éléments à créer et configurer, disons que vous étiez dans une architecture de couche (Layer Architecture) vous devrez décider en combien de couche vous allez utiliser. Les concepts d'architectures ne donnent pas des prescriptions à suivre, donc le détail spécifique à votre architecture doit être défini par l'architecte. Après l'instanciation des éléments, l'architecte doit allouer les responsabilités de chaque élément dans l'architecture ; par exemple, pour une application web, ceci prend la forme d'architecture de trois couches :

- **Couche de présentation** : la responsabilité est l'interaction utilisateur.
- **Couche de logique business** : la responsabilité est la gestion des activités de l'entreprise.
- **Couche de donnée** : la responsabilité est la persistance des données.

Enfin, l'architecte doit définir les interactions entre ces éléments, cela par l'élaboration des interfaces (Contrats de communication) qui définissent quelle information devrait circuler et sous quelle forme.

4.3.6 Étape 6 – « Capturer les vues et les prises de décision » :

À cette étape on a fini les activités de conception pour cette itération, cependant on doit capturer tous les vues (Sketch, diagramme, listings…) conçues dans les étapes précédentes de cette itération. Pour capturer les prises de décision on peut suivre le modèle suivant : « **<Un pilote>** est une priorité, alors on choisit le design **<Design concept>,** en acceptons le compromis **<Trade-off>** ». Cette étape permettra d'avoir une documentation préliminaire qui va aider dans les itérations ultérieures pour la prise de décision et pour améliorer progressivement la documentation.

4.3.7 Étape 7 « Analyser le design actuel et réviser l'objectif d'incrément et l'achèvement d'objectif de conception »

Au moment où vous atteignez cette étape, vous devriez avoir une conception partielle qui répond à l'objectif de l'itération. Vous pouvez effectuer des analyses sur les vues et les décisions, bien qu'il soit préférable de demander à quelqu'un d'autre de le faire. Avoir des feedbacks d'une autre partie est crucial pour vous aider à voir ce que vous pourriez avoir oublié ou mal fait. Une fois que vous avez analysé vos résultats d'itération, vous devez vérifier l'état de votre architecture et déterminer si elle satisfait l'objectif d'itération et si vous avez atteint un nombre d'itérations suffisantes pour satisfaire le but de la conception.

5. Conclusion

Dans ce chapitre, on a présenté l'architecture des systèmes d'information et ses procédures en générale, comme on a définis le rôle de l'architecte des systèmes dans une organisation. Puis on a entamé la méthode ADD 3.0 qui est la méthode choisis pour notre cas d'étude, où on a discuté sur sa procédure itérative, et comment chaque itération est piloté par des entrées tel que les attributs de qualité, les exigences fonctionnelles et aussi les contraintes. Notre préférence pour cette méthode se base sur le fait que la méthode est flexible et peut être adapté pour les systèmes Greenfield 4 et Brownfield 5.

De plus elle permet de documenter l'architecture et la prise de décision dans chaque itération. Enfin, sa nature itérative nous permet d'avoir une architecture initiale rapidement qui est idéal pour notre environnement de travail agile.

Chapitre II
Concepts d'architecture

1. Introduction

L'architecte fait face à plusieurs problèmes techniques compliqués, heureusement avec une immense communauté d'architecture des systèmes d'information, plusieurs produits se créent chaque jour, et la plupart de ces problèmes de conception ne sont pas aussi étranges et ont des solutions connues appelées **pattern**, **concept** ou **modèle**. Chacun de ces modèles favorise un **attribut de qualité** par rapport à un autre, donc bien choisir le concept ou les concepts à utiliser est crucial pour la réussite de l'architecture.

2. Layer Pattern « Modèle de couches » [32]

Ce concept est de loin le plus utilisé dans les systèmes d'information modernes, où il permet une **séparation de responsabilité** très convenable dans les systèmes avec plusieurs collaborateurs où le code du système est partitionné en modules distincts. Cela minimise le couplage du système et le rend plus facile à faire évoluer ou maintenir.

Point Fort	Point Faible
Maintenabilité Portabilité Testabilité Facile à implémenter	Chaque nouvelle couche va introduire une nouvelle abstraction et peut devenir très complexe et nuire à la performance du système en gros.

Tableau 3 Concepts d'architecture - ***Les points forts et les points faibles du Layer Pattern***

Exemple : Structure de projet typique

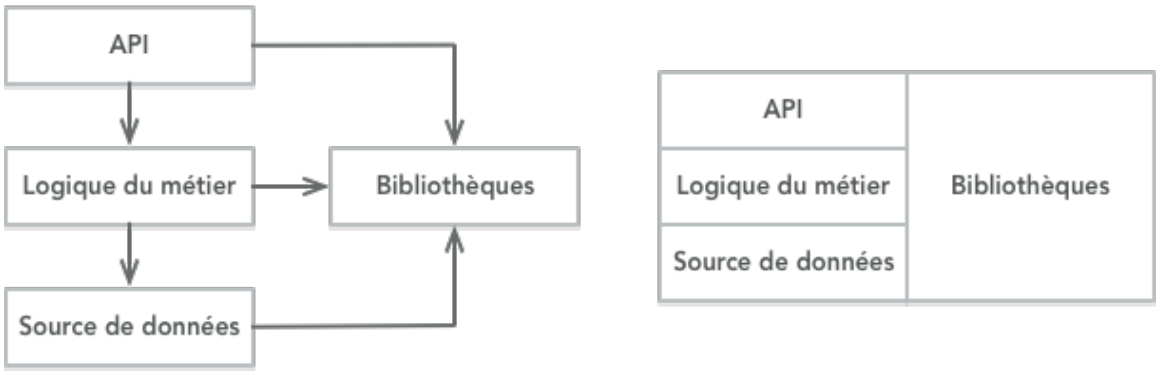

Figure 5 ***Architecture par couches***

3. Pipe-and-Filter Pattern

C'est un fameux modèle ou chaque composant effectue une transformation ou opération sur les données qui flux dans le pipeline. On appelle ces composants les filtres, ces derniers peuvent être arrangés dans plusieurs combinaisons pour avoir diffèrent résultats.

Exemple : Ligne de commande UNIX, démontre le modèle pipe-and-filter

```
sort data.txt | uniq
```

Figure 6 ***Ligne de commande UNIX***

Point Fort	Point Faible
Performance Portabilité Modificabilité	Les systèmes qui utilisent ce modèle sont généralement sans interactivité ou interface utilisateur.

Tableau 4 Concepts d'architecture - ***Les points forts et les points faibles du Pipe-and-filter Pattern***

4. Architecture des services « Service-Oriented Architecture »

Une approche assez intéressante où chaque composant du système est intégré comme un service indépendant qui satisfait un rôle précis. Les autres composants ignorent l'implémentation du service avec lequel ils communiquent. Généralement cette communication est via des verbes HTTP pour envoyer des requêtes et avoir des réponses.

Dans les organisations complexes ils optent pour une telle architecture pour laisser le libre choix aux différents départements d'implémenter leurs services, ou ils doivent juste respecter les normes de structure des réponses de leur **API**s pour permettre la communication avec les autres services. Un autre point fondamental pour que cette architecture marche est qu'il faudra que ces services aient un moyen de trouver le service à contacter, donc cette architecture devra implémenter un **Service Registry** qui actera comme un guide de coordination entre les services.

Exemple : Architecture Micro-services

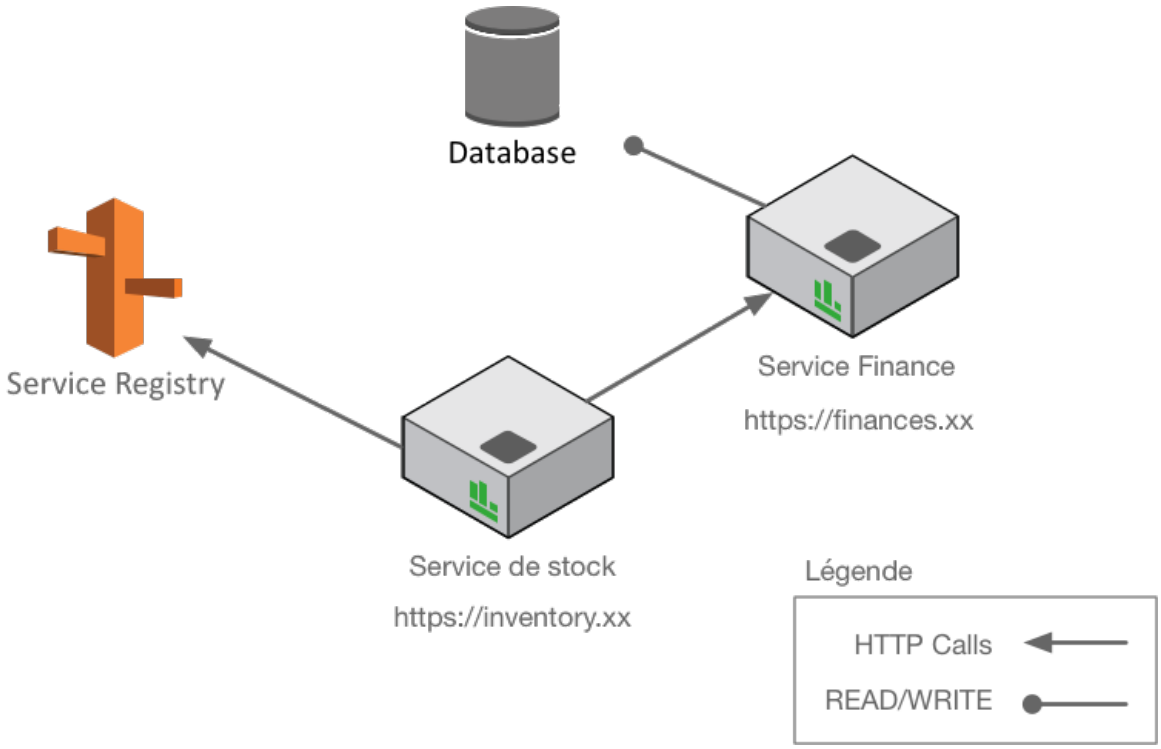

Figure 7 ***Architecture des micro-services***

Point Fort	Point Faible
Evolutif Agnostique à l'implémentation en interne Agnostique aux sources de données	Compliqué à déployer et de maintenir sa fiabilité.

Tableau 5 Concepts d'architecture - ***Les points forts et les points faibles de l'Architecture des services***

5. Architecture des étages (nivaux) « N-Tiers Architecture »

C'est un modèle largement utilisé, particulièrement dans les applications web. Il organise les structures architecturales en groupe logique qui réside dans des composants physiques tels qu'un serveur ou plateforme cloud. Cette architecture est similaire à celle des couches (Layer Pattern) avec la différence que N-Tiers organise les structures en **Runtime** en temps d'exécution alors que l'autre organise les modules en **Design-time** en conception.

5.1 Attributs [33]

- **Séparation de responsabilité** : Niveau de présentation, logique du métier et manipulation de données.
- **Communication synchrone** : La communication entre les niveaux est synchrone requête-réponse, les requêtes flux du niveau client vers les bas niveaux.
- **Facilite de déploiement** : Elle permet une liberté dans le déploiement où on peut déployer tous les niveaux dans une seule machine ou bien chaque niveau dans sa propre machine.

5.2 Exemple

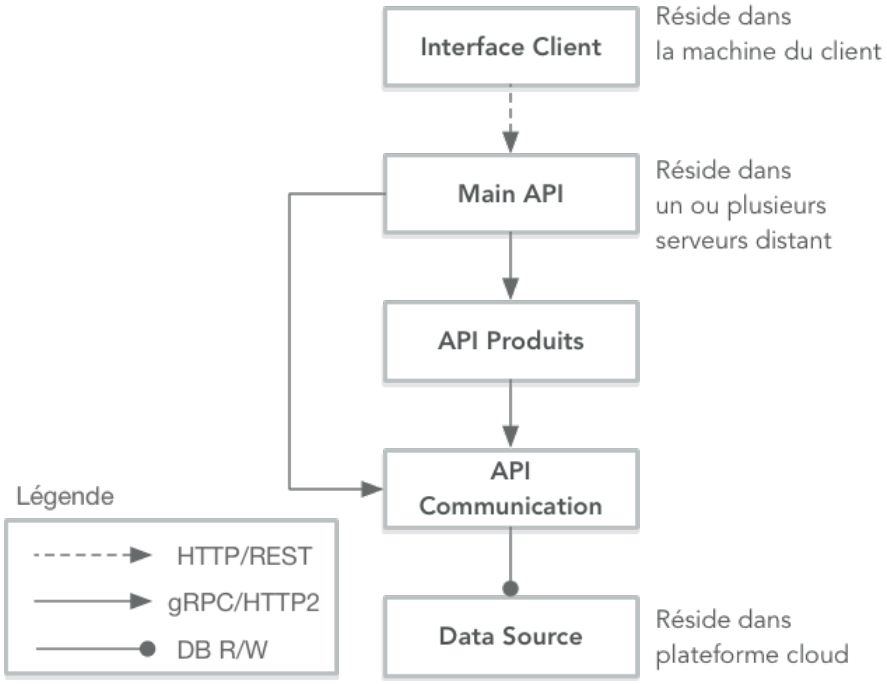

Figure 8 ***Architecture N-Tiers par nivaux***[32]

Point Fort	Point Faible
Sécurité Disponibilité Maintenabilité	Plusieurs niveaux peuvent nuire à la performance.

Tableau 6 Concepts d'architecture - ***Les points forts et les points faibles de l'Architecture N-Tiers par niveaux***

6. Big Ball of Mud

Ce n'est pas vraiment un modèle mais un phénomène qui arrive au moment où on ne suit aucun modèle en particulier. Il ne favorise aucun attribut de qualité, nuit à la maintenabilité, sécurité et extensibilité. En tant qu'architecte, il faut se méfier de ce phénomène car il peut émerger même avec un modèle choisis, ses modules ou composants sont enclins au risque d'être une boule de boue.

Ce phénomène arrive souvent dans les architectures micro-services où chaque service peut facilement devenir une boule de boue. Le seul point positive qu'on peut attribuer à ce phénomène est qu'il favorise la livraison rapide mais avec des dettes techniques et une perte d'intégrité du design.

7. Conclusion

Dans ce chapitre on a connu quelques patterns d'architecture, mais ce n'est qu'une goutte dans l'océan. De nombreux patterns sont publiés dans la littérature de l'architecture du software. S'il y a une règle à suivre, c'est qu'il n'y a pas un modèle miracle et en tant qu'architecte, on doit toujours voir l'architecture qui satisfait le mieux nos exigences.

Chapitre III
Modèles de travail

1. Introduction

Après avoir choisi un modèle d'architecture et l'avoir implémenté complètement ou partiellement, vient une pièce importante pour l'évolution du projet qui est la méthode de travail et de gestion du projet. La gestion des projets est un long séjour, avec des changements éventuels car les projets des systèmes d'information connaissent des fluctuations constantes. Dans les projets orienté planning, le plan du projet est un document fait au préalable qui indique ce que les développeurs ont à faire, comment le faire et en combien de temps ils doivent le réaliser. En plus des plans extensibles, on trouve aussi comment modifier le document du plan. Ce document est généralement structuré de la façon suivante : [33]

1- Introduction sur le projet et son objectif
2- Organisation d'équipe
3- L'analyse des risques
4- Exigences matérielles et humaine
5- Tâches et estimations
6- Planning
7- Suivi

La raison de jumeler la méthode de travail avec la gestion du projet est qu'ils sont étroitement liés. D'un côté le planning est extensif et très détaillé dans un environnement **Waterfall,** de l'autre on peut avoir une seule page comme plan de projet dans un environnement **Agile**.

2. Waterfall[34]

Approche plus linéaire aussi appelée **Plan-Driven,** elle peut prendre la séquence suivante :

1- Collecte et documentation des exigences.
2- Conception.
3- Développement et test unitaires.

4- Test d'intégration.

5- Test d'utilisateur facile.

6- Réparation et amélioration.

7- Livraison du produit finale.

Généralement, et dans un environnement Waterfall chaque étape représente une phase du processus du développement, chaque phase ne commence pas avant l'achèvement de la précédente.

Point Fort	Point Faible
• Accord mutuel entre le groupe de développement et le client sur le produit à livrer tôt. Ce qui simplifie la planification. • Mesure du progrès est plus simple car le projet est plus détaillé. • A part pour la revue ou validation la présence du client n'est pas nécessaire après la phase des exigences.	• Dans la réalité la collecte des exigences est rarement complète. • On risque que le client ne soit pas satisfait du livrable dans la phase finale.

Tableau 7 Modèles de travail - ***Les points forts et les points faibles du Waterfall***

3. Agile[10]

C'est un processus itératif, approche de développement orienté-équipe cette approche favorise la livraison rapide de produit fonctionnelle de façon itérative. Contrairement au mode Waterfall, il y a plus de planning détaillé en préalable. L'environnement agile généralement suit le concept du Time-Boxing autrement appelé Sprint dans la méthode Scrum, il s'agit d'une période close d'une semaine ou deux avec l'objectif d'arriver aux livrables fixés dans le sprint planning. Après l'achèvement du sprint, l'équipe fait un rétrospectif et évaluation de ce qui a été réalisé puis avoir une validation client sur le livrable. Dans un environnement agile l'interaction client est constante au long du projet. On parlera dans les deux sections qui suivent sur deux méthodes agiles utilisées dans l'industrie du développement de logiciel.

Point Fort	Point Faible
• Voir le résultat rapidement et fréquemment, permet d'amortir l'impact des décisions majeures. • Donner au client le sens de contribution et de possession sur le projet. • Être le premier au marché.	• L'équipe doit maîtriser la bonne communication avec le client. • Tous les membres doivent être actifs sur le même projet.

Tableau 8 Modèles de travail - ***Les points forts et les points faibles de la méthode Agile***

3.1.Scrum[34]

Le Scrum est un Framework d'organisation et de développement. Son principe est de découper les tâches en sprint à un rythme constant avec des livraisons très fréquentes. Cette méthode fournit un cadre de gestion de projet avec des rôles, des réunions, des artefacts, des règles de gestion et un cycle itératif de développement. Son objectif phare est d'améliorer la productivité des équipes, tout en permettant une optimisation du produit grâce à des feedbacks réguliers du marché. Pour mieux organiser le travail cette méthode définit 3 rôles importants qui sont :

- **Le « Product Owner » :** il porte la vision du produit à réaliser, il est chargé de remplir le backlog et de déterminer la priorité des user stories à réaliser.
- **Le « Scrum Master » :** Il a pour responsabilité, dans le cadre du développement d'un produit, d'aider l'équipe à travailler de façon autonome et à s'améliorer constamment. Il est le garant de l'application du processus, Scrum en l'occurrence.
- **L'équipe de développement** : elle est chargée de transformer les besoins exprimés par le product owner en fonctionnalités réelles, opérationnelles et utilisables.

La méthode Scrum se caractérise par une répartition de chacune des tâches à faire. L'équipe trie les fonctionnalités et tâches qu'elle réparti dans des Sprints qui est définit comme suit :

- **Sprint :** Un projet est organisé autour de « sprints » de développement qui est un intervalle de temps pendant lesquels l'équipe va compléter un certain nombre de tâches du backlog d'une durée allant généralement de deux à quatre semaines.

3.2. **Kanban** [35]

Kanban est une méthode largement utilisée dans les équipes de développement agile. D'origine japonaise instituée par Toyota depuis 50 ans. Inspirée de la méthode de gestion de stock des supermarchés où ils exposent juste la quantité nécessaire pour satisfaire la demande, pour optimiser le flux entre le supermarché et les consommateurs.

5.1 Kanban comme méthode de développement

Kanban est une méthode qui se concentre principalement sur la création ou amélioration des produits. Elle permet de contrôler et visualiser le flux de développement en toute transparence.

Structure : la structure de Kanban est une structure flexible. Chaque organisation a avec son implémentation particulière du kanban. Kanban est simplement un tableau avec des colonnes où chaque colonne représente une étape du processus de développement ; Généralement on trouve une colonne appelée **Backlog** où toutes les **tâches**, **idées**, et **suggestions** résident sous forme de **post-it**. Suivi par une colonne de **Spécification**, dans cette colonne les tâches sont spécifiées et découpées en sous-tâches équitables avec leur spécification pour qu'ils soient implémentées. Ce qui nous ramène à la colonne **Implémentation** c'est la colonne qui démontre ce qui est en train d'être implémenté et qui est entrain de l'implémenter. Et enfin vient la dernière colonne de **Validation** après que les tâches aient été implémentées ils passent à la validation qui est la phase finale du processus.

Figure 9 ***Notre Implémentation Physique Kanban***

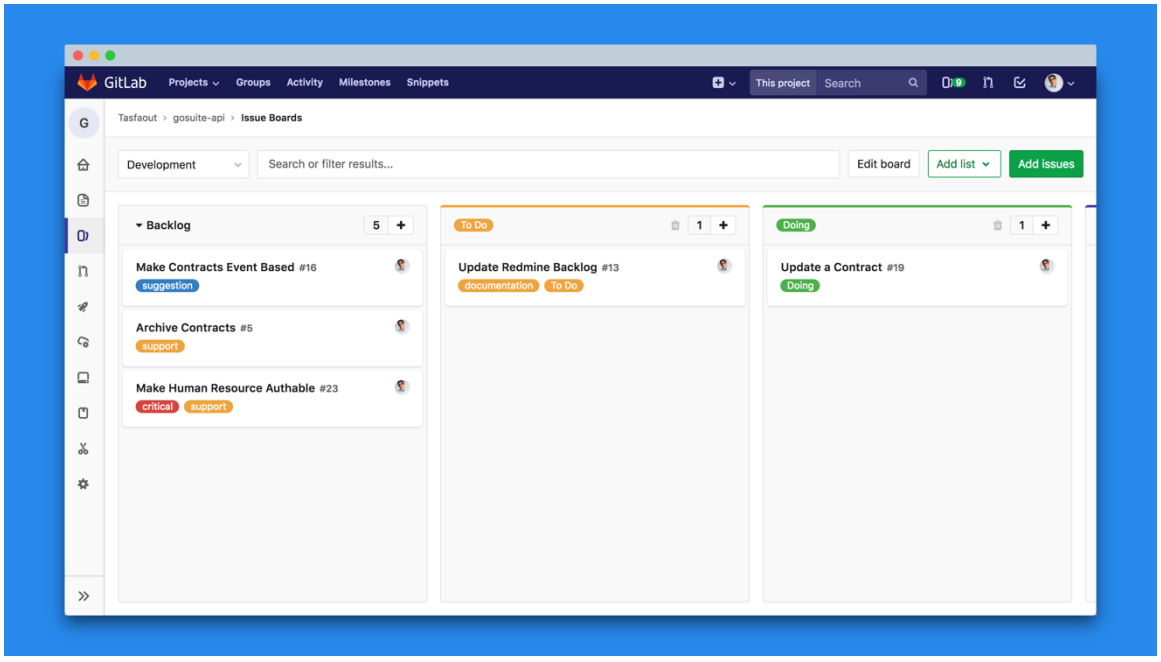

Figure 10 ***Kanban Interface Web***

3.2.1. Implémentation du Kanban

Durant notre travail on a pu implémenter la méthode Kanban dans notre flux de travail. Et on a apporté des adaptations pour plusieurs types de projets ; uni-projet ou multi-projet. Car kanban est par défaut adapté envers les projets où toute l'équipe travaille sur un seul projet. Pour implémenter Kanban dans votre espace de travail vous avez le choix entre **les plateformes digital** ou bien un **tableau physique**, nous avons opté pour le dernier car il est plus visuel et on a toujours une idée sur le flux et ses blocages. Quel que soit votre choix, Kanban a quelques impositions :

- Définition de limite **WIP** (Work in Progress) le nombre de tâches qui peuvent être traité par phase.
- Définir le critère d'une tâche résolue, pour maintenir la qualité du travail dans chaque étape du **Kanban**.
- Faire les **Standups**[10] chaque journée.

Étapes d'implémentation:

a. **Création du tableau** : On a suivi un modèle fait par Microsoft Xbox Team[35]

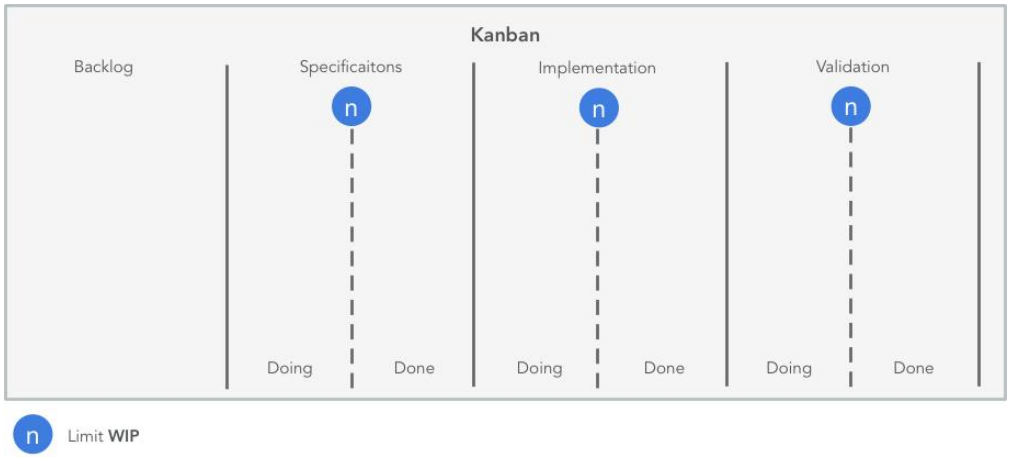

Figure 11 ***Modèle Kanban***

[10] Des brefs meetings pour savoir ce qui a été accompli, détecter d'éventuels blocages.

- Backlog : Encapsule toutes les tâches aussi appelées Stories, ces dernières peuvent être des fonctionnalités, des suggestions ou bien des bugs[11]. Chaque post-it représente une story, ils sont ordonnés par priorité de haut au bas (Top-down) les post-it peuvent être réordonnés par l'équipe à tout moment.
- Spécifications : Permet de voir les tâches choisies du Backlog qui sont en train d'être découpées dans la première sous-colonne (Doing) et les tâches découpées prêtes à être implémentées (Done).
- Implémentation : Permet de voir les tâches qui sont en train d'être implémentées en production dans la première sous-colonne et les tâches prêtes à être validées dans la seconde colonne.
- Validation : Contient les tâches à valider et tâches livrées pour validation externe.

Le flux Kanban

Figure 12 ***Flux Kanban***

b. Définir le WIP

Cette étape est cruciale dans le processus Kanban, où il faut mettre en place un WIP limite pour chaque colonne du Kanban. Il s'agit tout simplement du nombre de post-it permis dans une étape (colonne) donnée. Cette limitation permet de faciliter la transition en cas de changement soudain de priorité, protéger les membres d'équipe et aussi à s'adapter à la vitesse de l'étape la plus lente du processus. Il y a plusieurs méthodes pour définir cette limite, comme la méthode proposée par Eric Brechner de Microsoft que le nombre à donner au WIP n'est pas évidant au départ et que cette limite doit être encline au changement en faisant des benchmarks sur les capacités de l'équipe. Néanmoins, comme point de départ les groupes agiles définissent cet WIP en prenant le nombre de membre dans chaque étape plus 50% de tampon.

[11] Toute défaillance ou erreur inattendu dans un programme.

Exemple : pour un groupe de deux développeurs le WIP de la phase d'implémentation

$$\mathbf{WIP_{Implementation} = 2 + (2) * 50\% = 3}$$

Puis le responsable sur l'équipe ajustera le WIP selon le débit de productivité dans la sortie.

c. Définir le critère des tâches résolues

C'est une particularité du kanban par rapport aux autres méthodes agile, elle institue un mécanisme pour maintenir la qualité du travail réalisé. Il s'agit d'imposer entre équipe une checklist d'assurance qualité avant de dire qu'une tâche est résolue. Généralement, avant de mettre une tâche en résolu elle doit être testée avec des tests unitaires et des tests d'intégration. Ce qui rend Kanban idéal pour la programmation en TDD[12].

d. Standups

Les Standups se font généralement chaque matin et ne doivent pas dépasser les 5-15 minutes. Ces réunions sont menées par le kanban master ou bien un membre sénior. Les points qui doivent être abordés dans le standup :

- Demander aux membres de l'équipe s'il y a un blocage ou besoin d'assistance dans une tâche donnée.
- Célébrer les progrès de l'équipe.
- Récapituler avec l'équipe le progrès de chaque étape du kanban.

[12] Test-Driven Development

4. Les bonnes pratiques du développement[36]

Le manque d'évolutivité aperçu dans les systèmes d'information, est un souci généralement causé par la violation des principes de conception primordiale des systèmes d'information. Bien que ces principes soient plus abstraits, ils sont les composants de tout système existant. Leurs correcte implémentation est la fondation pour tout système évolutif. Les principes présentés dans cette section sont en relation avec la programmation orienté-objet, quoiqu'ils restent abstraits pour être implémentés dans d'autres styles de programmation.

"Make things as simple as possible, but no simpler." –Albert Einstein

4.1. Simplicité

La simplicité est un concept très relatif où il n'y a aucune norme de mesure pour cette dernière, un développeur doit toujours se demander est que ce choix est simple pour moi ou pour le client ? Est-ce que je dois procéder à une implémentation simple ou à une autre, simple à maintenir dans le futur ? La simplicité ne veut pas dire que la solution soit raccourcie ou rapide mais plutôt à quel degré de difficulté notre solution sera à comprendre ou à maintenir pour nous-mêmes ou d'autres développeurs dans le futur. Dans ce chapitre on parlera de quatre modèles basiques pour promouvoir la simplicité dans les produits du software.

4.2. Cacher la complexité et créer des abstractions

Avec l'évolution d'un système on commence à perdre la vue globale, car il aura de nombreux détails qu'il sera difficile de tous les retenir. Pour permettre la compréhension du système à tout moment, l'abstraction de la complexité des composants nous permet de faire des zooms en avant et en arrière avec facilité. Cependant, il n'est pas toujours évident d'abstraire ou même de savoir quoi abstraire, il est toujours plus simple de mettre en place des abstractions dans un système Greenfield en visant la simplicité locale dans chaque situation de développement. Dans la simplicité locale, il s'agit de s'assurer que lorsqu'on analyse un module ou une class on peut rapidement comprendre ses responsabilités sans

avoir à connaître les détails sur les parties distantes. La simplicité locale permet de voir les modules avec une vue de haut-niveau et savoir leur responsabilités sans savoir comment ils sont implémentés. Considérer cet exemple

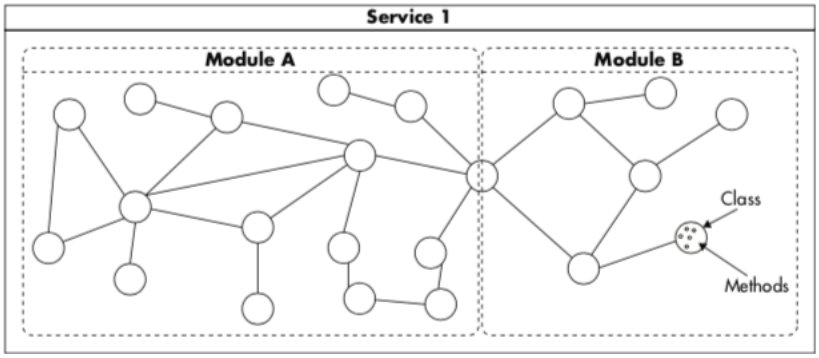

Figure 9 ***Les Niveaux d'abstraction***[36]

Les points représentent les méthodes de niveau le plus bas d'abstraction puis les nœuds démontrent les classes ou interface le second niveau d'abstraction, les liens symbolisent les dépendances entre les classes et interface. Puis il y a les modules un autre niveau d'abstraction où leurs responsabilités est figurée dans des interfaces publiques comme notre exemple où il y a une seule interface publique entre **Module A** et **Module B**. Puis un autre niveau d'abstraction qui est le **Service** qui devient nécessaire si le système atteint une haute complexité.

4.3. Éviter les architectures non-nécessaires

L'une des pratiques pour promouvoir la simplicité, est d'éviter d'ajouter des complications non-nécessaires. Parfois en essayant de prédire des cas d'utilisation spéciaux, nous perdons la concentration sur les cas d'utilisation primaires. Et on se trouve à résoudre tout problème imaginable, qui rend notre système complexe de façon inutile. Un système bien conçu permet la facilité d'ajout des détails et des fonctionnalités dans le futur, pour arriver à un tel système on commence avec un degré raisonnable d'abstraction puis on incrémente dessus. Cela résulte en un meilleur système, au lieu d'essayer de prédire tout cas d'utilisation ou fonctionnalité qui peut être ou ne pas être nécessaire dans le futur.

4.4. Adopter le TDD

C'est une pratique qui a prouvé être la plus efficace non seulement pour promouvoir la simplicité mais aussi un outil d'assurance de qualité. C'est une pratique de développement orienté-tests où les ingénieurs écrivent les tests avant d'écrire le code de la fonctionnalité. C'est une approche radicale mais avec des grands bénéfices, car elle force le développeur à implémenter ce qui est nécessaire pour la fonctionnalité pour valider le test. Les tests servent aussi comme une bonne documentation du code pour l'équipe.

Exemple d'un test unitaire de notre cas d'étude :

```php
public function it_should_add_a_secondary_tenant_to_resource()
  {
    //Given
    // A Physical Resource That has a rent contract
    // Start Date & End Date
    $startDate = new Carbon('08/05/2018 00:00');
    $endDate = new Carbon('12/05/2018 12:00');
    //When
    $contract = $this->physicalResource->rent($this->tenant, $startDate, $endDate);
    // Adding an other tenant
    $otherTenant = factory(HumanResource::class)->create();
    $this->physicalResource->addTenant($otherTenant);
    //Then
    $this->assertCount(2, $this->physicalResource->getTenants());
  }
```

Figure 14 ***Exemple de test unitaire en PHPUnit***

4.5. Apprendre des leaders de l'industrie

L'un des bénéfices de faire partie d'une large communauté de développeurs c'est qu'on peut utiliser les outils des autres et on peut voir leurs propres patterns vers des programmes compréhensibles et évolutifs. Vous pouvez explorer les APIs et Framework suivants :

- **Laravel** : Un Framework web PHP open-source avec un architecture MVC très propre, nous l'avons utilisé dans notre cas d'étude à cause de sa simplicité et de la facilité d'extension. Au cours du développement avec ce Framework vous constaterez qu'il a été pensé pour satisfaire tous les besoins d'une application web moderne. De plus vous pouvez explorer son code source, c'est l'un des codes les mieux lisibles dans la communauté.
- **Google Maps API** : Un très bon exemple des APIs flexibles qui résout des problèmes complexes et d'une façon extrêmement simple. Son intégration est très facile une carte avec un marqueur peut prendre moins d'une heure et cela avec le temps de la création de la clé d'API. Comme vous pouvez creuser encore avec la superposition des revêtements, modifier l'interface d'interaction et le calcul des distances compliqués.

5. Le principe SOLID[36]

Un principe de programmation orienté objet instauré par ***Robert C. Martin*** connu par le surnom de **Uncle Bob.** Où il s'agit de cinq concepts de programmation :

- Single Responsibility Principle

  Un concept efficace pour réduire la complexité du code, il prêche qu'un class doit avoir une seule responsabilité pas plus. Ce concept réduit l'accrochage des classes, augmente la simplicité et la facilité de les maintenir et les faire évoluer.

- Open/Closed Principle

  "Good architecture maximizes the number of decisions not made". –Robert C. Martin

  C'est à propos de la création du code qui n'a pas besoin d'être modifié quand les exigences changent ou un nouveau cas d'utilisation arrive. Open/Closed veut dire « *Open for extension and closed for modification* » " Ouvert pour extension et fermé pour la modification ". Ce principe permet d'avoir toujours de la flexibilité pour les décisions sur les détails inconnus ou ambigües.

- **Exemple**

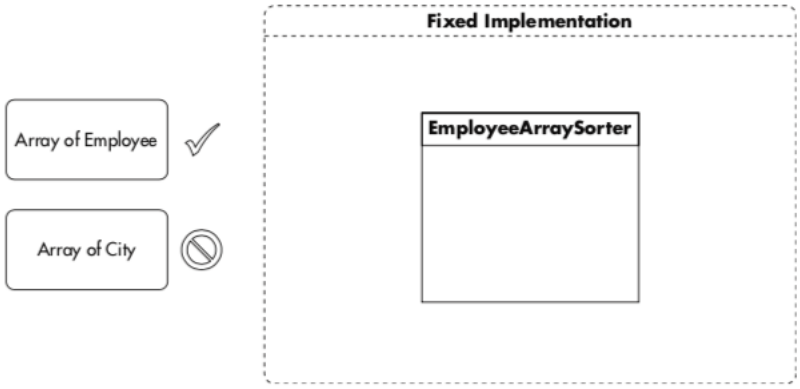

Figure 15 ***Implémentation d'un trieur d'employés sans le principe Open/Closed***

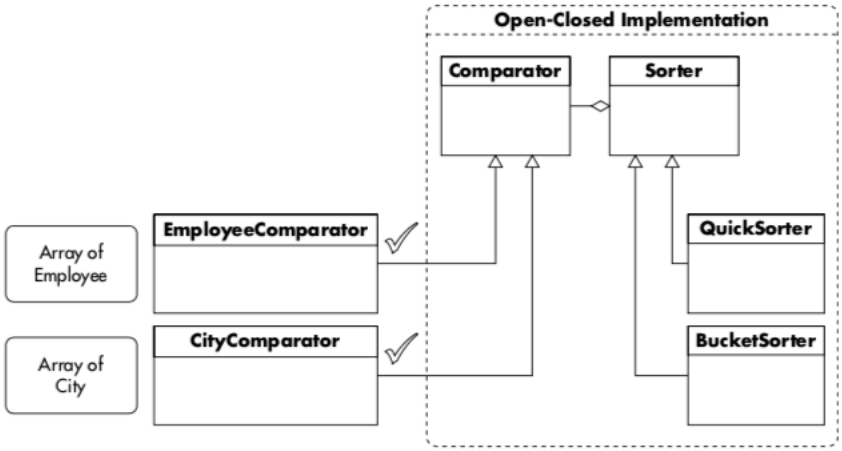

Figure 16 ***Implémentation avec le principe Open/Closed***[1]

- Liskov Substitution Principle

  Cet principe dit que les objets doivent être remplaçables par leur subtypes[13] sans causer la défaillance de l'objet. Dans l'exemple qui suit on voit comment la class **Driver** peut implémenter les SubTypes **Car** et **ElectricBus.**

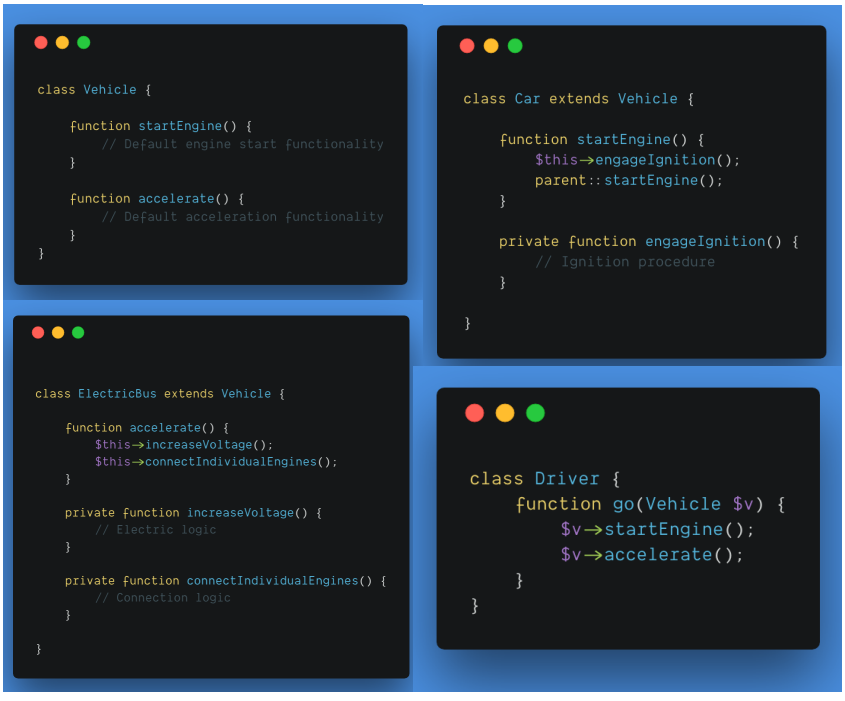

Figure 10 ***Exemple de Liskov Substitution Principle***

[13] Les interfaces que l'objet implémente.

- Interface Segregation Principle

  Avoir plusieurs interfaces spécifiques est toujours mieux qu'une seule interface qui représente plusieurs méthodes spécifiques. Cela imposera sur les classes qui implémentent cette dernière d'implémenter toutes ces méthodes et cela cause la violation du principe du **Single Responsability**

- Dependency Inversion Principle

  Dans notre cas d'étude pour les extensions, on implémente ce principe où dans l'injection des dépendances des classes fils, on utilise une interface et non une implémentation (Classe concrète). Cela nous permet d'avoir plusieurs implémentations si l'implémentation de base ne satisfait pas les exigences. Ou pour la truquer dans les tests unitaires

```php
namespace App\Access\Terminals;

use App\Access\Terminals\ITerminal;
use App\Access\Traits\PhysicalResourceExtension;
use App\Base\Resources\Physical\Contracts\IPhysicalResource;
use Illuminate\Support\Facades\DB;


class Terminal implements ITerminal
{
  use PhysicalResourceExtension;
  protected $interface;
  // Dependecy Injection of the PhysicalModule via it's interface
  public function __construct(IPhysicalResource $PRModule)
  {
    $this->PRModule = $PRModule;
  }
  // Dependecy Injection of the TerminalModule via it's interface
  public function setInterface(ITerminal $interface)
  {
    $this->interface = $interface;
    return $this->interface;
  }
```

Figure 18 ***Injection via les interfaces : Class responsable de la gestion du terminal d'accès avec deux injections ; une pour le module de ressource physique et l'autre pour l'interface de contrôle du terminal.***

6. Conclusion

Le concept derrière toutes les méthodes agiles, c'est le flux itératif. Il permet de livrer les produits d'une façon incrémentale et de voir le progrès d'évolution du produit final et de l'équipe. Cependant, agile n'est pas une réponse pour toutes les organisations ou tout type de projet. Les méthodes agiles fonctionnent très bien dans les équipes multidisciplinaires et les projets complexes avec une grande imprévisibilité. Pour les projets prévisibles et répétitifs où on peut créer des modèles de déploiement, les méthodes **Plan-Driven** sont nettement mieux et plus rentables.

En tant que développeur ou architecte de système, votre rôle est de fournir des solutions convenables pour votre entreprise sous des contraintes de temps et d'argent, il faut avoir l'esprit ouvert à d'autres points de vue. Il n'est pas toujours vrai que la solution la plus propre est la plus appropriée. Il faut rester réaliste et procéder à résoudre les problèmes avec une mentalité pragmatique, en ayant à l'esprit qu'il n'y a pas de meilleure façon de faire, il y a plusieurs meilleurs façons de faire selon le contexte.

Chapitre IV
IMPLÈMENTATION D'ÉTUDE DE CAS.

1. Déploiement du ADD

Dans ce chapitre on établit l'implémentation du ADD 3.0 sur notre étude de cas. Où on va détailler extensivement sur le processus de définition de l'architecture et l'implémentation de la totalité du système du back-end au front-end.

1.1 Business Case & Motivation

Dans le but de développer une interface de gestion d'accès pour les établissements, qui aide à gérer l'affectation d'accès aux employés dans une institution, on a décidé avec la collaboration de «**NetSprint »** de créer une architecture de système qui nous permettra d'arriver à un degré d'extensibilité pour non seulement pouvoir créer un système de gestion de contrôle d'accès mais pouvoir l'exploiter en d'autres systèmes dans d'autres domaines tel que la gestion d'hôtellerie, la gestion de stock et commercialisation. En outre, on doit satisfaire les scénarios attributs de qualité suivants :

- Extensibilité : le système doit être capable d'être étendu de manière simple sans influencer sur le système de base.
- Performance : les interactions du système en interne et avec ses interfaces publiques doivent être fluide ; < 1s pour une requête simple.
- Convivialité : l'interface de gestion d'accès doit être simple à utiliser, en minimisant le nombre de clics pour arriver à une action définie.
- Scalabilité : La structure de base de la base de données ne doit pas être modifiée, elle doit satisfaire tous les besoins définis tout en gardant un degré d'extensibilité.

Pour arriver à créer l'interface de gestion de contrôle, dans notre cas d'étude on a comme première distribution le support des terminaux d'accès de la marque **ZkTeco**. Comme contrainte on doit utiliser leur propre bibliothèque compatible **C#**, puisque nous utilisons **PHP** comme langage de serveur, on doit aussi créer un micro service **RPC**[14] pour permettre la communication entre l'API **RESTFul**[15] principale et les terminaux qui sont installés

[14] Remote Procedure Call - Type d'API ou les requêtes représentent des actions et non des ressources.
[15] Representational State Transfer

dans l'institution comme le montre le diagramme de contexte *figure 5*. Pour arriver à un tel système on l'a divisé en sous-systèmes ; un système de base qui gère nos cas d'utilisation d'une façon abstraite, le système de gestion d'accès qui étend depuis le système de base comme illustré dans le diagramme d'architecture de référence *figure 6*, puis le système intermédiaire pour les communications avec les terminaux **ZkTeco**.

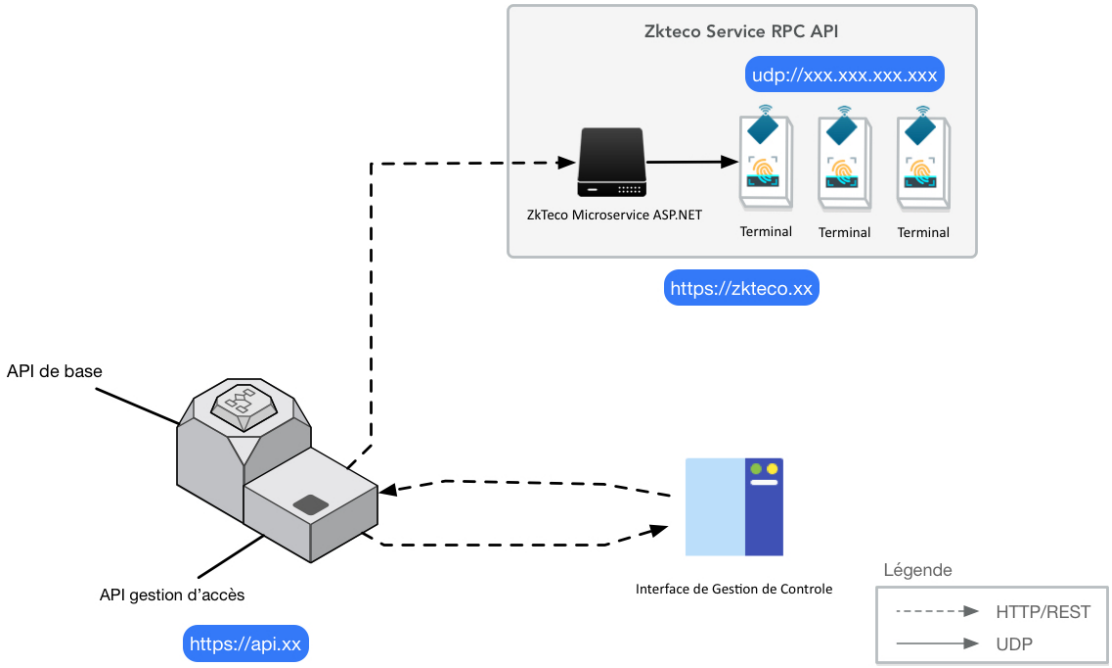

Figure 11 ***Diagramme de contexte pour notre système***

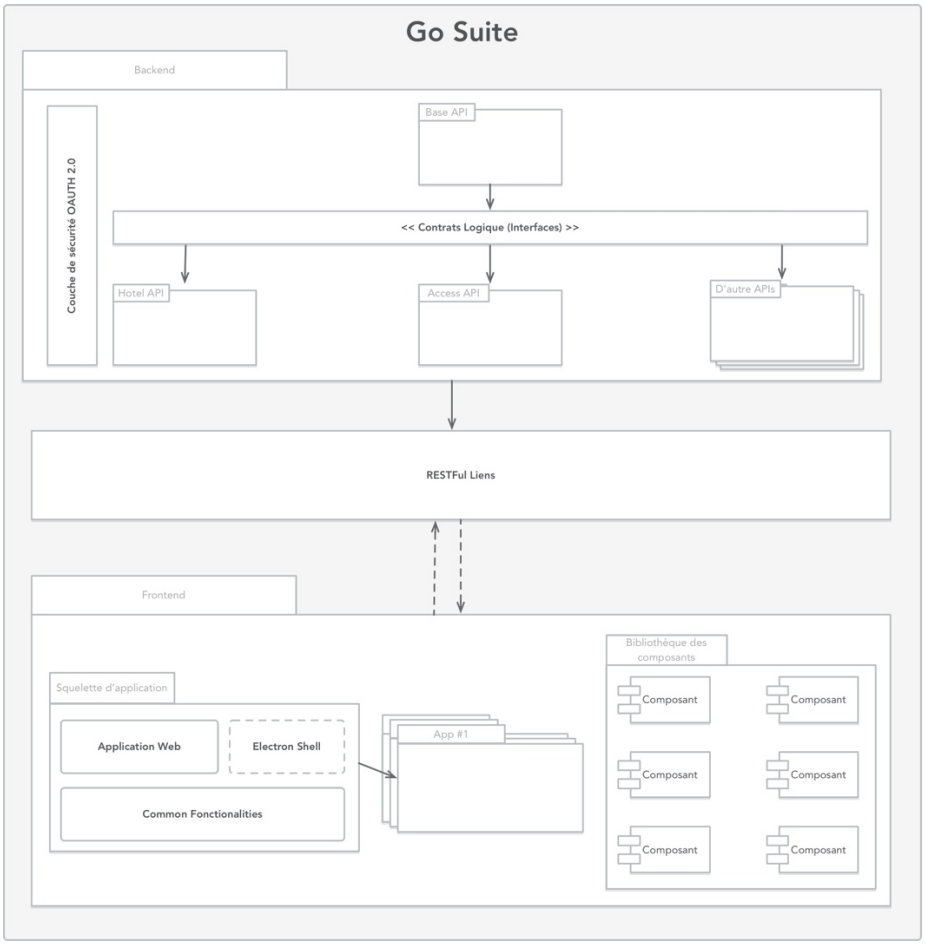

Figure 12 ***Architecture de Référence***

2. System de base (Pilotes)

2.1 Les cas d'utilisation

Le tableau suivant représente les cas d'utilisation de l'API de base, qui est l'une des entrées de notre itération ADD

Cas d'utilisation	Description
UC-1-B : Étendre sur les fonctionnalités d'acquisition de base	Permettre au développeur de créer de nouvelles fonctionnalités de réservation, de location, ou même d'achat. Avec la stratégie de base de louer ou acheter une ou plusieurs **ressources physiques** pour une seule

	ressource humaine résultant en un **contrat de location** ou **contrat d'achat**.
UC-2-B : Créer un arbre des ressources physiques ou un objet physique et étendre sa fonctionnalité.	Permettre au développeur **deux façons** de représenter ses **ressources physiques** dans son application d'extension, la première possibilité est de les représenter en **arbre hiérarchique**, et la deuxième est de les représenter avec une **entité indépendante**. Le développeur a le libre choix de définir des **propriétés** de définition de la **ressource physique.**
UC-3-B : Créer une ressource humaine, client, employé, ou entreprise**.**	Permettre au développeur de représenter ses **ressources humaines** dans son application d'extension, soit une représentation de personne générique, client, employée ou entreprise. Le développeur a le libre choix de définir des **propriétés** de définition de la **ressource.**
UC-4-B : Créer des rapports et des factures avec la possibilité d'extension	Permettre au développeur de générer des **factures** ou **des rapports** avec un **contenu, en-tête, marge et bas de page** définie par le développeur. Dans le cas de facture le développeur doit définir la **maquette** de la **facture** et la **recette de calcul**.
UC-5-B : Authentification	Permettre au développeur d'intégrer un système d'authentification fourni par le **Framework Laravel**
UC-6-B : Autorisations	Permettre au développeur d'intégrer un système d'autorisation fourni par le **Framework Laravel**

Tableau 9 Étude de cas system de base - ***Cas d'utilisation de l'API de base***

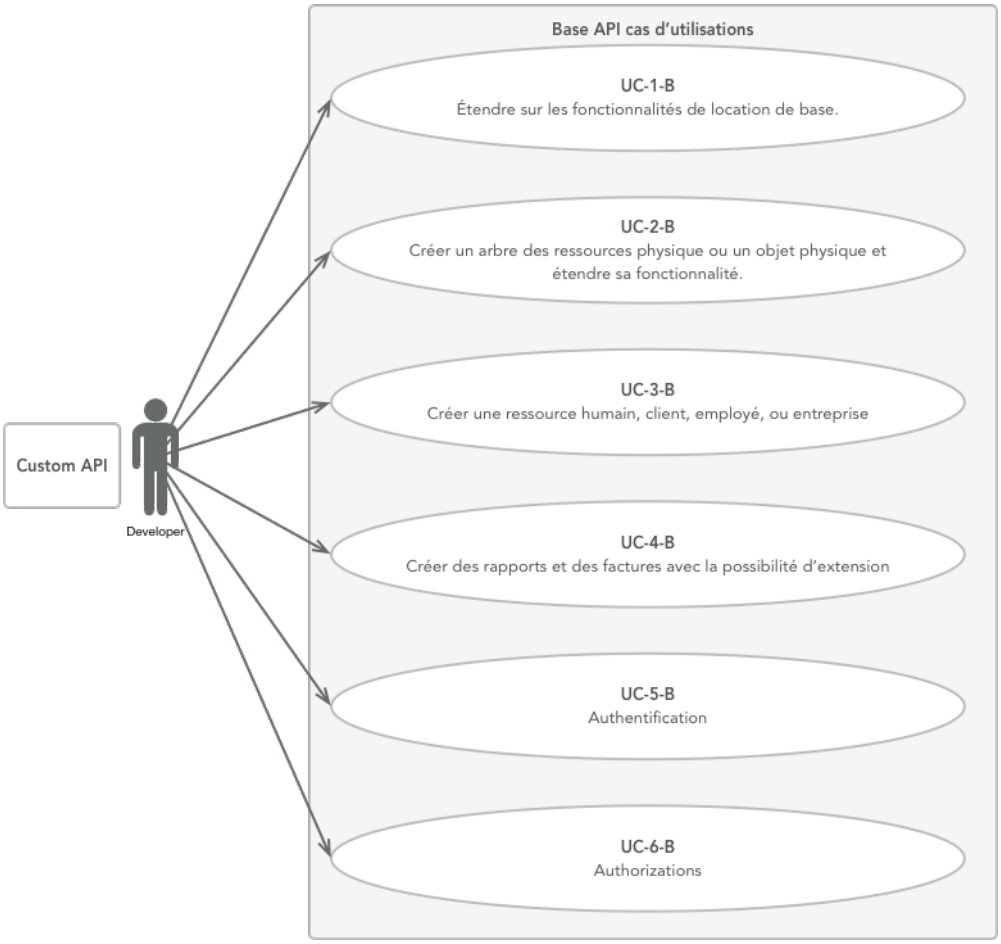

Figure 13 ***Les cas d'utilisation de l'API de base***

2.2 Attributs de qualité

Attribut de qualité	**Scénario**	**Cas d'utilisation associé**
QA-1-B : Extensibilité	Le développeur n'a qu'à injecter la classe du module primaire pour avoir ces méthodes et pouvoir les étendre dans sa propre classe.	**UC-1-B**
QA-2-B : Performance	Les interactions doivent être fluides, avec des mécanismes de renvoi en cas d'échec.	/
QA-3-B : Scalabilité	La structure de base de la BDD ne doit pas être modifiée, mais avoir la possibilité d'étendre ses instances	**UC-1-B**

*Tableau 10 Étude de cas system de base -****Attributs de qualité pour le système de base***

3. System de gestion d'accès (Pilotes)

3.1 Les cas d'utilisation

Cas d'utilisation	Description
UC-1-A	Permettre aux **utilisateurs** de voir l'état des **terminaux** d'accès qui contrôlent les **serrures** avec une organisation par **immeuble, étage, section, porte.** Les états sont groupés par catégorie : **1. Communication (avec le terminal)** : a. Online b. Offline **2. Communication (avec la porte) :** a. Ouverte b. Fermée 3. **Action** : a. En Lecture b. Ouvert c. Fermé
UC-2-A	Permettre aux **utilisateurs** qui s'occupent de la gestion d'accès d'ajouter les codes **RFID** des employées, à **des terminaux spécifiques, zone(s), étage(s), immeuble(s),** ou tout **l'établissement**.
UC-3-A	Permettre aux **utilisateurs** qui s'occupent de la gestion d'accès d'ajouter des **politiques de restrictions** sur **un employé ou un groupe d'employés** par **zone**.
UC-4-A	Permettre aux **utilisateurs** qui s'occupent de la gestion d'accès d'ajouter des **politiques de restrictions** sur **un employé ou un groupe d'employés** par **période de temps**.
UC-5-A	Permettre aux **utilisateurs** qui s'occupent de la gestion des employés **d'ajouter des employés, créer des groupes** et **gérer la liste des employés.**
UC-6-A	Permettre aux **administrateurs** de **visualiser l'historique** d'accès d'un employé ou un groupe d'employés et **imprimer un rapport** qui contient **des détails chronologiques d'accès.**
UC-7-A	Permettre aux **super administrateurs** de **visualiser le journal d'utilisation** de l'application avec le schéma suivant : **<user> a <action> le <date time>**
UC-8-A	Permettre aux **administrateurs** de créer des privilèges et les attribuer à des **profils d'utilisateurs**.

*Tableau 11 Étude de cas system de gestion d'accès -**Cas d'utilisation de l'interface de gestion d'accès***

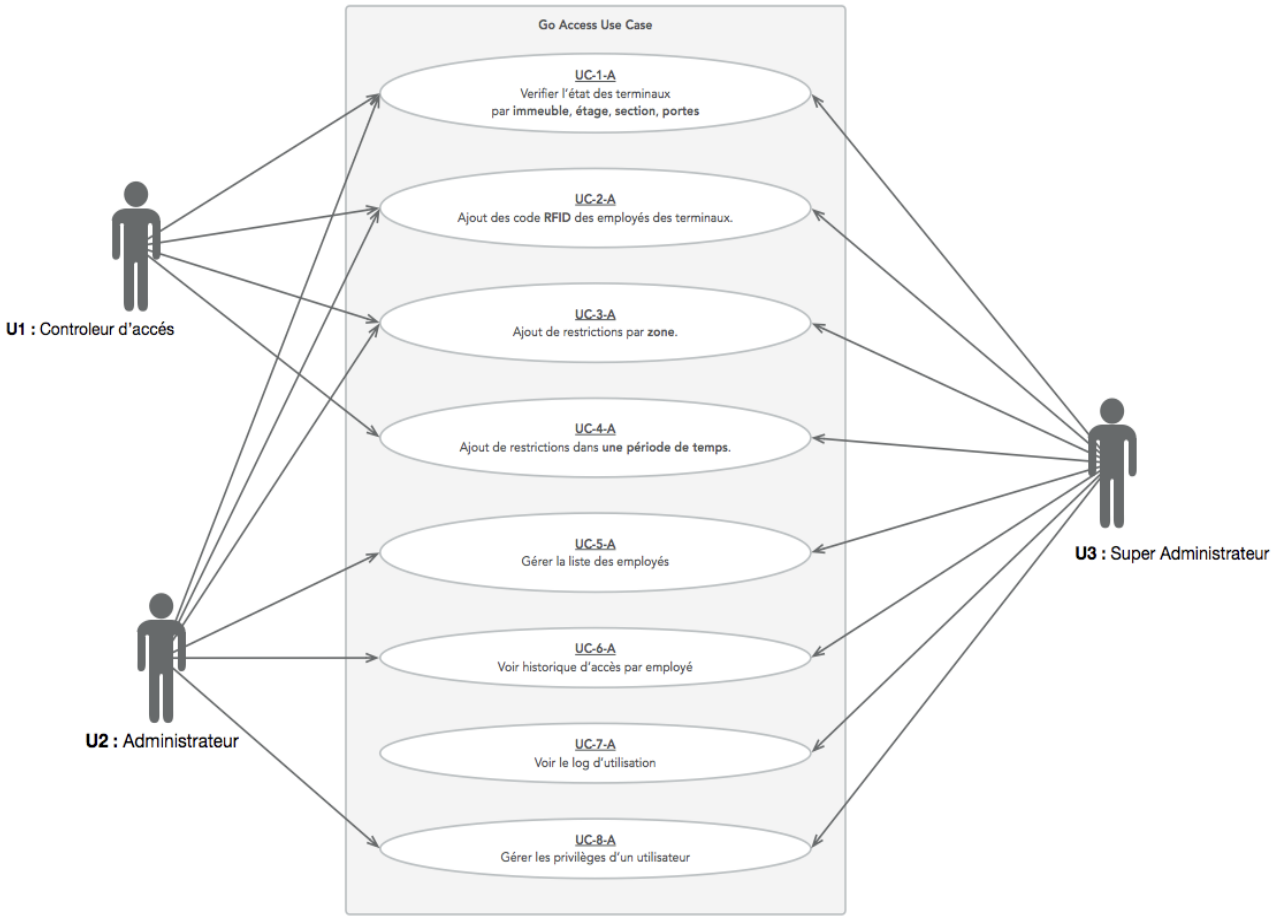

Figure 14 ***Cas d'utilisation d'interface de gestion d'accès***

3.2 Attributs de qualité

Attribut de qualité	**Scenario**	**Cas d'utilisation associé**
QA-1-A : Convivialité	L'interface de gestion doit être simple à utiliser, et fluide.	/
QA-2-A : Précision	L'état des terminaux doit être précis et en temps-réel.	**UC- 1-A**

Tableau 12 Étude de cas system de gestion d'accès - ***Attributs de qualité pour le système de gestion d'accès***

3.3 Contraintes

ID	Détails
CON-1-A	La bibliothèque **DLL** de communication avec les terminaux **ZkTeco** est développé en **C#**
CON-2-A	La communication avec les terminaux est en **UDP** qui peut nuire à la précision et cause la perte des paquets

Tableau 13 Étude de cas système de gestion d'accès - ***Contraintes pour le système de gestion d'accès***

3.4 Soucis Architecturales

ID	Détails
CRN-1-A	Le service de communication avec les terminaux ajout un micro-service en **ASP.NET C#** qui est une système à maintenir indépendamment. De plus l'équipe est expérimenté en **PHP**.
CON-2-A	Avoir un niveau supplémentaire de communications avec les terminaux peut nuire à la performance sous forme de latence sure les réponses.

Tableau 14 Étude de cas système de gestion d'accès - ***Soucis Architecturales pour le système de gestion d'accès***

4. Implémentation du système de base

4.1 ADD Étape 1 : Révision des pilotes

Comme discuté dans le premier chapitre sur le ADD, la première étape est de revoir les entrées pour choisir quels sont les pilotes à supporter dans cette itération. Dans cette itération on va supporter les fonctionnalités primaires du système de base.

Catégorie	Détails
Objectif de conception	Créer un système Greenfield[16] , qui va servir de système de base évolutif extensible. Dans le domaine de la gestion des ressources physique et humaine.
Fonctionnalités Primaires	Pour cette itération on va supporter suffisamment de cas d'utilisation pour permettre le system de gestion d'accès d'être implémenter dans la seconde itération. • UC-1-B • UC-2-B • UC-3-B • UC-5-B
Attributs de qualité	• QA-1-B • QA-3-B
Contraintes	/
Soucis Architectural	/
Architecture Existante	/

Tableau 15 Étude de cas système de base - ***Sélection des pilotes d'entrées***

4.2 Itération 1 : Étape 2 : Objectif de l'itération

Dans cette itération on va voir comment on va implémenter des cas d'utilisations **UC- 1-B, UC-2-B, UC-3-B, et UC-5-B.** De plus on doit maintenir l'attribut de qualité **QA-1-B, et QA-3-B**

[16] System non-existant

4.3 **Étape 3 :** Choisir les éléments du système à affiner

Vue que c'est un système Greenfield et c'est la première itération, donc nous sommes en train d'affiner la totalité du système.

4.4 **Étape 4 :** Choisir les concepts d'architecture pour satisfaire les entrées sélectionnées

Choix Architectural	**Rationalisation**
Appliquer un pattern modulaire.	Chaque cas d'utilisation traite un domaine spécifique, en suivant la bonne pratique de séparer les responsabilités. On constitue notre architecture de trois modules principaux : • Module d'acquisition : Gère les contrats de location et d'achat des ressources physiques affectées à des ressources humaines. **UC- 1-B** • Module de ressource physique : Gère les ressources physiques d'une façon hiérarchique ou singulières. Et leur liaison avec les contrats d'acquisition. **UC-2-B** • Module de ressource humaine : Gère les ressources humaines, et leur liaison avec les contrats d'acquisition. **UC-3-B**
Les modules implémentent un ORM	Chaque module est une classe avec son interface, la classe hérite d'un ORM[17] qui va aider à simplifier les interactions de la base de données dans le module et ses extensions.
Utiliser Base de données relationnelle	Utiliser une base de données relationnelle pour la couche des données, Ils sont simples à implémenter et sécurisés par défaut. Bien qu'ils ne soient pas autant évolutifs que les bases de données non-relationnelles ils sont plus performants pour les opérations d'agrégation. Pour support **QA-3-B** pour l'extension des modules de ressource physique et humaine, on met en place un champ nommé « définition » de type **JSON**[18]. Cela facilitera l'extension des tables.

[17] Object Relational Mapping : Fais le lien entre la classe et la table dans la base de données

[18] Javascript Object Notation : Structure de donnée légère pour les communication web

Utiliser les interfaces pour chaque module	Les interfaces sont très utiles pour définir une abstraction de l'implémentation concrète. Elles nous permettent aussi l'extension des modules en utilisant l'injection de dépendance DI[19]. Cela pour support **QA-1-B.**
Utiliser le modèle TDD	Le modèle TDD permet une évolution prédictible du développement du système, facilite l'ajout des nouvelles fonctionnalités et maintenir la stabilité.[1]

Tableau 16 Étude de cas système de base - ***les concepts d'architecture choisis pour satisfaire les entrées sélectionnées***

4.5 **Étape 5 :** Instancier les éléments architecturales, allocation des responsabilités

Choix Architectural	**Rationalisation**
Héberger le projet dans un PAAS[20] avec CI Intégration continue	Ce choix nous permet d'avoir un Écosystème de développement et de déploiement continu. En travaillant avec un gestionnaire de versioning tel que **Git** avec système d'intégration continue **Gitlab** CI on peut garantir une distribution agile de nos itérations. Chaque téléchargement vers le serveur distant déclenchera notre suite de tests, si ces derniers passent avec succès, la version sera déployée automatiquement dans le PAAS pour la validation des clients ou partenaires.
Implémentation du protocole d'authentification OAUTH 2.0	Vu qu'on est en train de développer une **API RESTFul** il n'y a pas un moyen pour utiliser un système de session pour gérer les authentifications. Cependant le protocole **OAUTH** est très approprié pour les API où il utilise un mécanisme de « **Token** » qui expire dans une période de temps, ce dernier doit toujours être accompagné des requêtes client **HTTP** plus précisément de le « **Headers** » de la requête pour que le serveur exécute cette dernière et enfin donne une réponse. Cela pour supporter **UC-5-B.**

[19] Dependency Injection : Méthode performante et testable d'extension des classes dans la programmation orienté-objets

[20] Platform As A Service : Service cloud avec une interface interactive pour guider le déploiement d'un code source, de plus d'une gestion des diffèrent service d'hébergement.

Utiliser Laravel comme couche de base	Laravel offre autant d'outils qui peuvent accélérer le développement et éviter de réinventer la roue. Ce framework se base déjà sur le modèle MVC[21]. Dans notre situation on n'aura besoin que du MC car l'API n'a aucune couche de présentation, uniquement des réponses de données en JSON[22].
Configurer la base de données	Configurer deux bases de données MySQL; une pour production qui est de la version en instance, et un autre pour les tests unitaires. Celle-ci gardera l'intégrité des tests car après chaque test contre la base de données, les données sont supprimées pour ne pas erroner le test suivant.

Tableau 17 Étude de cas système de base - ***Instanciation des éléments architecturales, allocation des responsabilités***

4.6 Étape 6 : Capturer les diagrammes

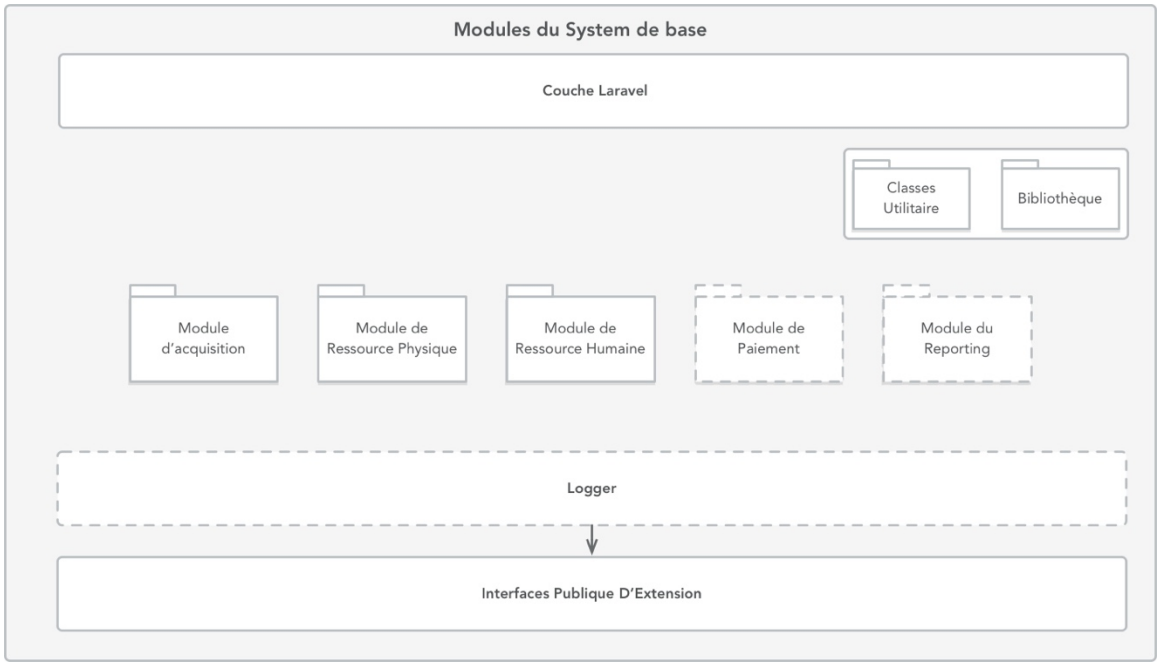

Figure 15 ***Architecture de référence du système de base***

Dans cette représentation modulaire de notre système, nous utilisons un modèle de couche. La couche de base est **Laravel** puis nos modules se superposent ensuite à une

[21] Model-View-Controller : Le modèle représente la couche donnée, view est la présentation, et le controller et le lien entre les deux.

[22] Javascript Object Notation : Structure de donnée nativement compatible avec javascript, contrairement au XML

couche Logger (System de Journal) qui suit les interactions des modules avec l'extérieur. Enfin, la sortie de notre système passe par des interfaces publiques.

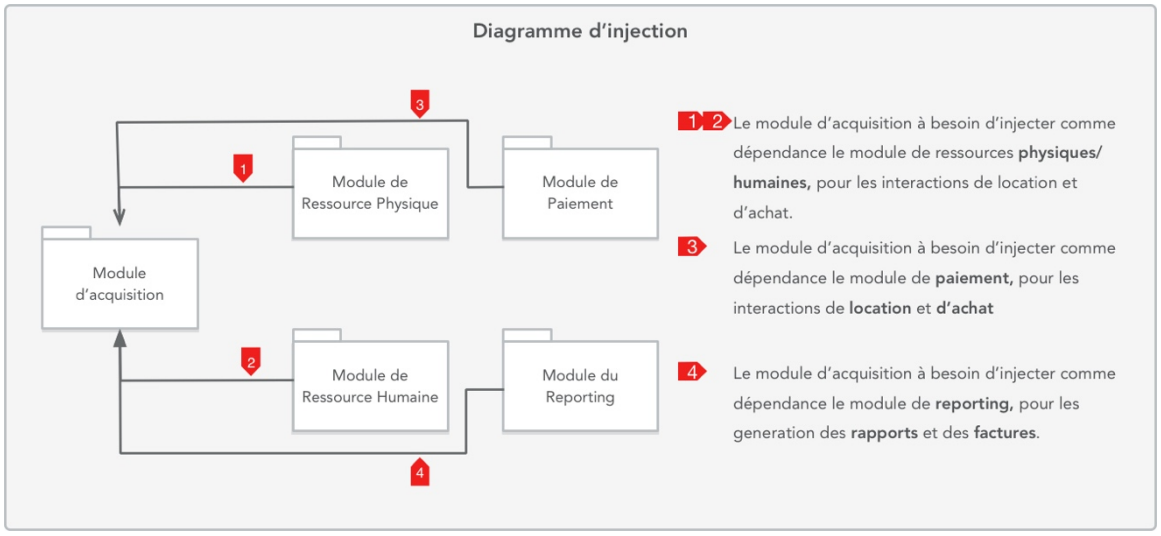

Figure 16 ***Diagramme qui représente les dépendances entre module***

4.6.1 **Diagramme de classes :** pour définir notre architecture des classes de chaque module, on a passé par deux phases ; réaliser un prototype pour prouver la validité de l'architecture puis l'intégrer avec **Laravel.**[23]

[23] PHP qui est un langage dynamique, donc les diagrammes utiliseront un typage dynamique.

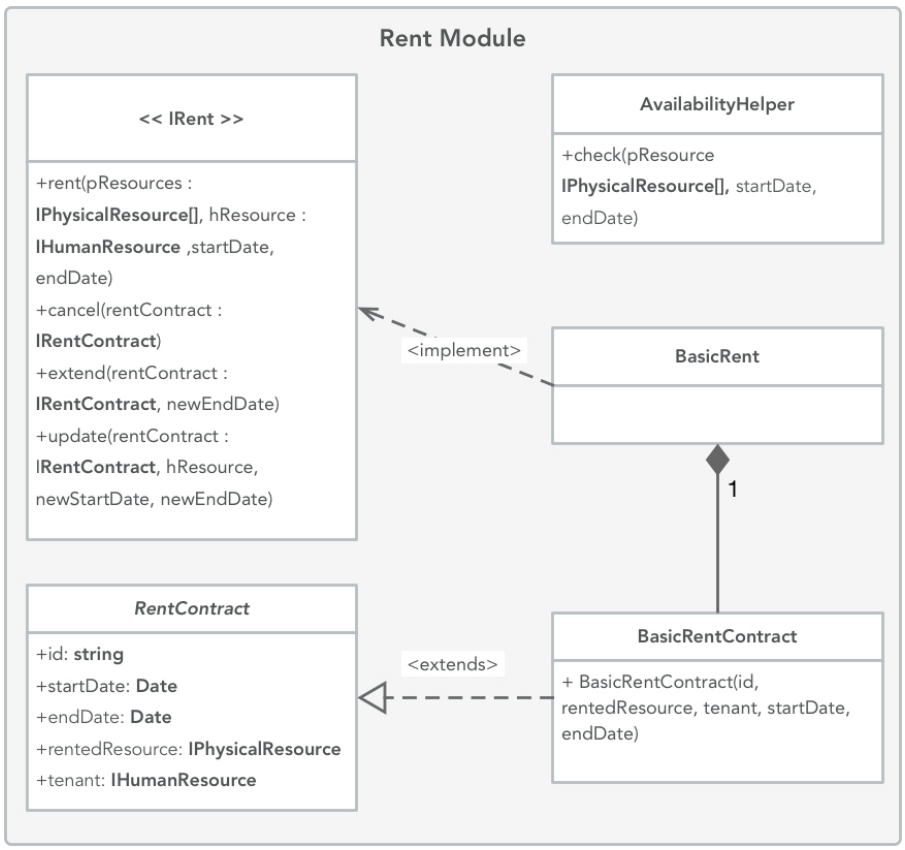

Figure 17 ***Digramme de classe prototype pour le module d'acquisition (Location) utilise Decorator [24] pattern***

Cette disposition de class suit la (**Decorator Pattern**) c'est une pattern qui permet décorer la class avec une autre via son constructeur. Disant qu'on veut implémenter une nouvelle stratégie de location **CustomRentPolicy** en dessus de la stratégie de base **BasicRent**, cette pattern nous permet de le faire de la façon suivante :

```
$customRentPolicy = new CustomRentPolicy(new BasicRent);
```

Figure 18 ***Decorator Pattern***

[24] Pattern de programmation orienté-objet

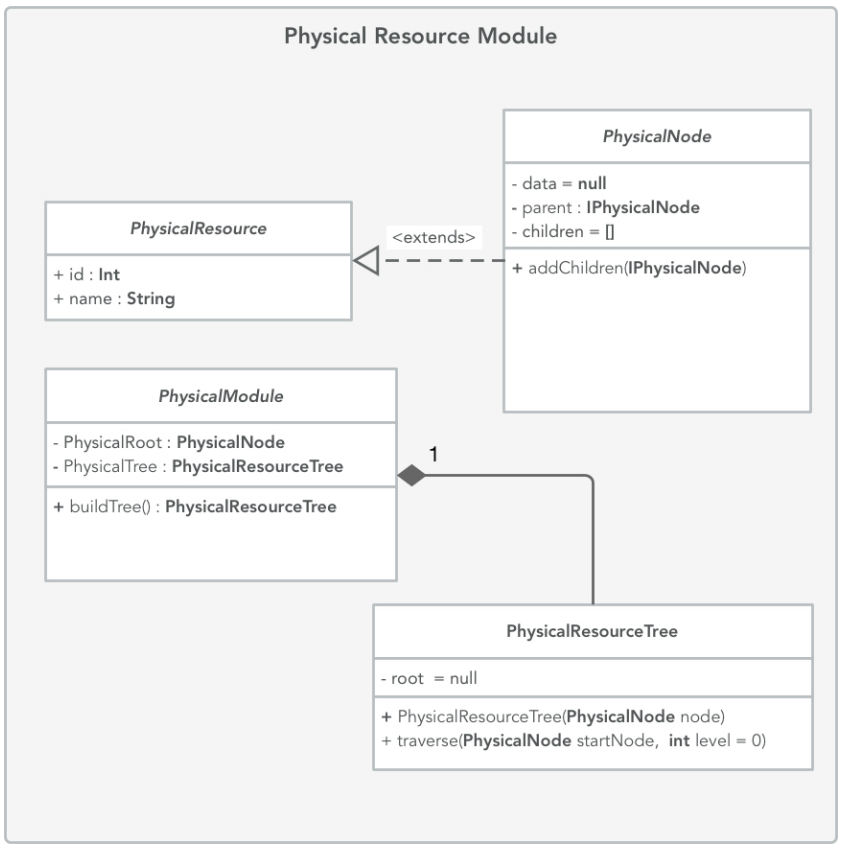

Figure 19 ***Diagramme de classe prototype du module de ressource physique***

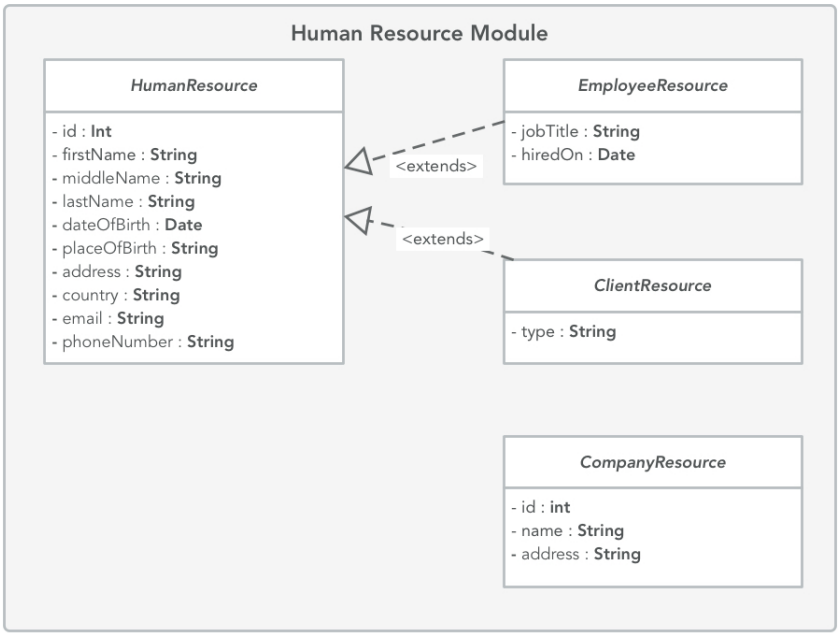

Figure 20 ***Diagramme de classe prototype du module de ressource humaine***

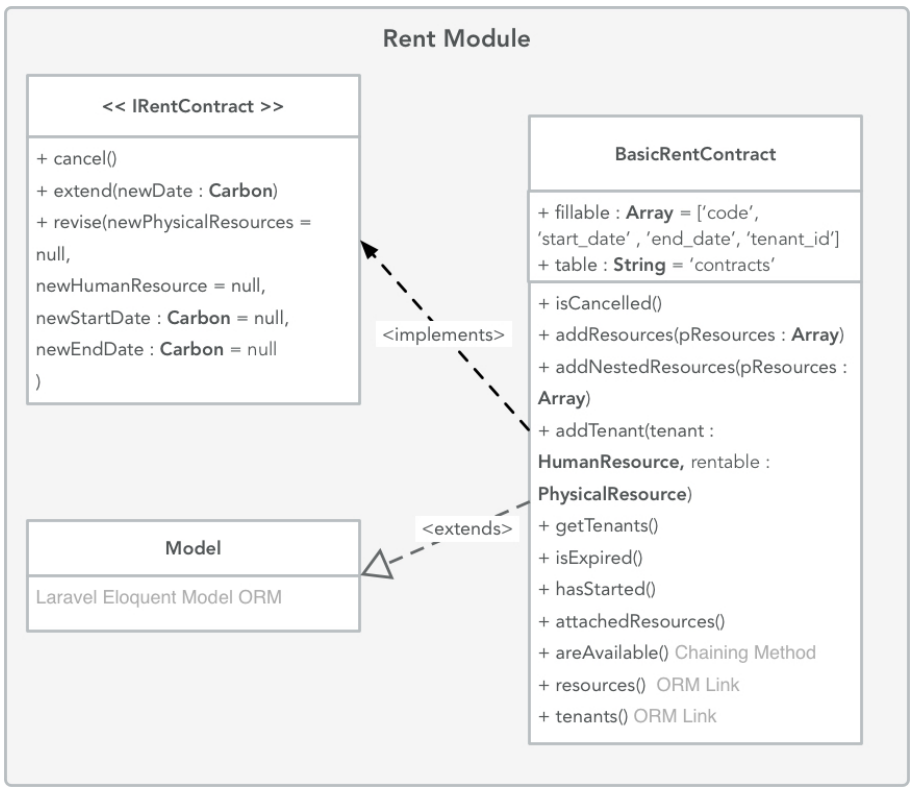

Figure 21 ***Digramme de classe implémenté avec Laravel***

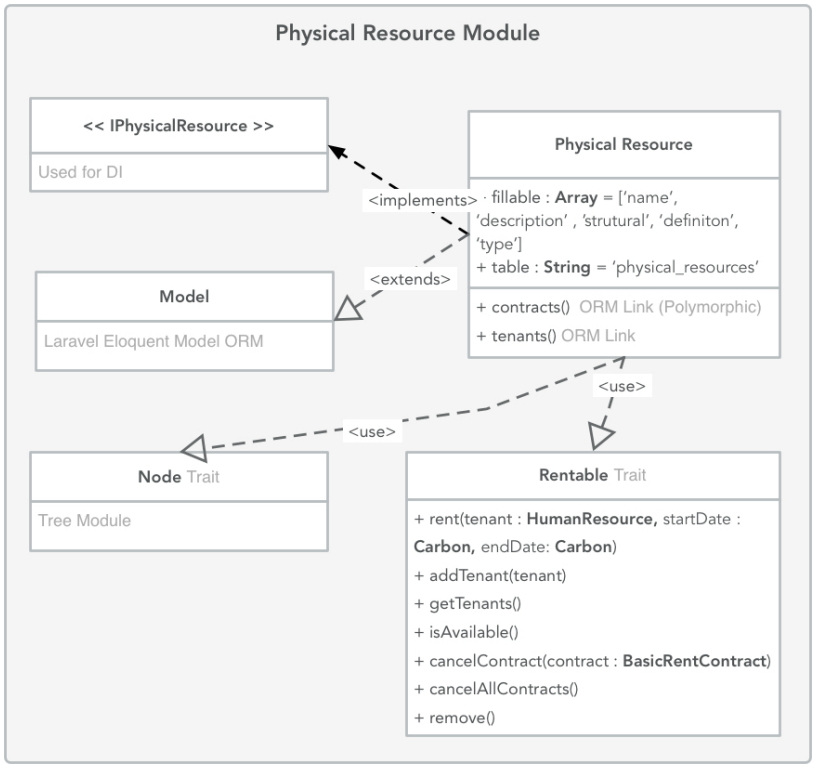

Figure 22 ***Digramme de classe du module de ressource physique avec Laravel***

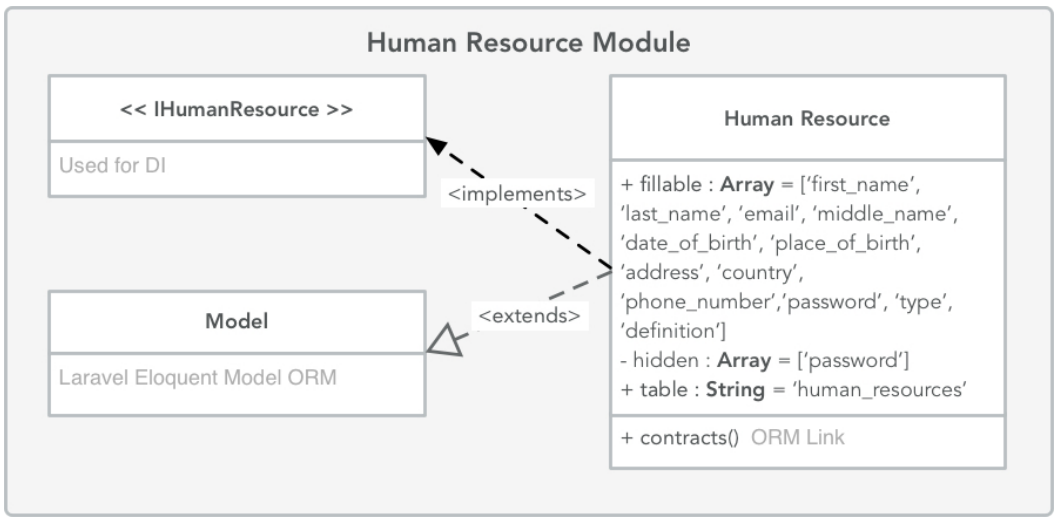

Figure 23 ***Diagramme de classe du module de ressource humaine***

4.7 Étape 7 : Analyser le design actuel et revoir l'objectif de l'itération et son achèvement Dans cette itération on a pu adresser les cas d'utilisation et attributs de qualité définies dans l'objectif de l'itération.

Non Traité	Traité Partiellement	Complètement Traité	Choix et décision de conception
UC-4-B		**UC-1-B**	Créer un module qui gère la création des contrats d'acquisition pour les ressources physiques assignées à des ressources humaines. L'extension de ce module se fait depuis une injection de dépendance avec l'interface **IRentContract** dans un nouveau module créé par le développeur tel qu'un module de réservation d'hôtel.
UC-6-B		**UC-2-B**	Le module des ressources physiques peut être étendu avec une injection de dépendance dans le module d'extension et avoir les fonctionnalités d'arborescence et aussi les fonctionnalités d'acquisition depuis la trait **Rentable**
		UC-3-B	Le module des ressources humaines gère ses manipulations, et

	l'extension se fait avec de l'injection de dépendance.
UC-5-B	Implémentation du protocole **OAUTH2.0** permet la protection de l'API et signe toute requête qui arrive.
QA-1-B	L'extension dans notre système utilise le principe de l'injection de dépendance, mais pour qu'elle marche il faudra un Service Container. Heureusement, **Laravel** contient un **Service Container**[25] qui va configurer toutes ces injections automatiquement lorsqu'on injectera nos dépendances dans les controllers et manuellement dans les autres cas.
QA-3-B	On utilise des champs en **JSON** dans les tables pour permettre l'extension des tables.

*Tableau 18 Étude de cas system de base - **Analyse du design actuel***

5. Implémentation du système de gestion d'accès

5.1 ADD Étape 1 : Révision des pilotes

Dans cette itération on va supporter les fonctionnalités primaires du système de gestion d'accès.

Catégorie	Détails
Objectif de conception	Créer une API RESTFul avec une interface client d'exploitation, qui va aider à gérer les accès, pointage et restrictions des employés dans un établissement.
Fonctionnalités Primaires	Pour cette itération on va supporter les fonctionnalités suivantes : • UC-1-A • UC-2-A • UC-3-A • UC-4-A

[25] Permet de résoudre les dépendances en Runtime, trouver les implémentations et les injecter.

	• UC-5-A • UC-6-A
Attributs de qualité	• QA-1-A
Contraintes	• CON-1-A
Soucis Architecturale	• CRN-1-A
Architecture Existante	/

*Tableau 19 Étude de cas système de gestion d'accès - **Sélection des pilotes d'entrées***

5.2 Itération 2 : Étape 2 : Objectif de l'itération

Dans cette itération on va voir comment on va implémenter des cas d'utilisations définis**.** De plus on doit maintenir les attributs de qualité avec les contraintes et résoudre les soucis architecturaux

5.3 Étape 3 : Choisir les éléments du system à affiner

Puisque les modules de base sont déjà implémentés, dans cette itération on procède à l'extension.

5.4 Étape 4 : Choisir les concepts d'architecture pour satisfaire les entrées sélectionnées

Choix Architectural	**Rationalisation**
Appliquer un pattern modulaire.	Chaque cas d'utilisation traite un domaine spécifique, en suivant la bonne pratique de séparer les responsabilités. On constitue notre architecture de cinq modules principaux : • Module gestion d'immeuble :**UC-1-A** • Module de restriction: **UC-3-A** et **UC-4-A** • Module gestion d'employés : **UC-5-A** • Module d'historique d'accès : **UC-6-A** • Module ZkTeco : **UC-2-A**, **UC-3-A** et **UC-4-A**
Interface client web	Avoir une interface client interactive web et adapté aux mobiles accessible à tout moment. **QA-1-A**

API RPC	Pour résoudre le **CON-1-A** et **CRN-1-A** on doit implémenter un micro-service web **RPC** en **C#** pour effectuer des opérations sur les terminaux **ZkTeco** on utilisant leur propre bibliothèque **DLL**

Tableau 20 Étude de cas system de gestion d'accès - ***Choix des concepts d'architecture pour satisfaire les pilotes choisis***

5.5 Étape 5 : Instancier les éléments architecturaux, allocation des responsabilités

Choix Architectural	Responsabilités
Implémentation du routage **REST** et les controllers de l'API de gestion d'accès	Le routage suivant la norme REST nous permet de donner une représentation intuitive des modules de notre application. Suivant un verbe http le routeur permet de dispatcher la requête vers la méthode appropriée dans un Controller donné puis vers le modèle pour faire une manipulation de données si nécessaire. Exemple : • **HTTP Verb** : 'GET', Path : /api/employees – Donne la liste de tous les employés. • **HTTP Verb** : 'GET', Path : /api/employees/{id} – Donne les détails d'un seul employé. **HTTP Verb** : 'POST, Path : /api/employees/, Data : {employeeData} – Créer un nouveau employé avec les données envoyées.
Implémentation de l'API **RPC** pour les communication avec **ZkTeco**	Création d'API en ASP.NET implémente la bibliothèque de **ZkTeco**. Les routages sont de type RPC pour clarifier les actions à envoyer vers l'API. De plus toutes les requêtes ont dans leur header l'adresse IP du terminal à commander. Exemple : • **HTTP Verb** : 'GET', Path : /terminal/users – Donne la liste de tous les employés ou utilisateurs avec leur carte RFID ou empreinte digitale.
Implémentation du front-end client	Création d'un front-end web[26] à partir du Framework **ReactJs** de **Facebook**. Permet la composition des éléments web d'une façon évolutive et portable car il suit une architecture orienté-composant. Chaque composant est un fichier **Javascript** qui contient la logique de présentation et interaction avec les autres composants.

Tableau 21 Étude de cas system de gestion d'accès - ***Instanciation des éléments architecturaux et allocation des responsabilités***

[26] Couche de présentation chez le client

5.6 **Étape 6 :** Capturer les diagrammes

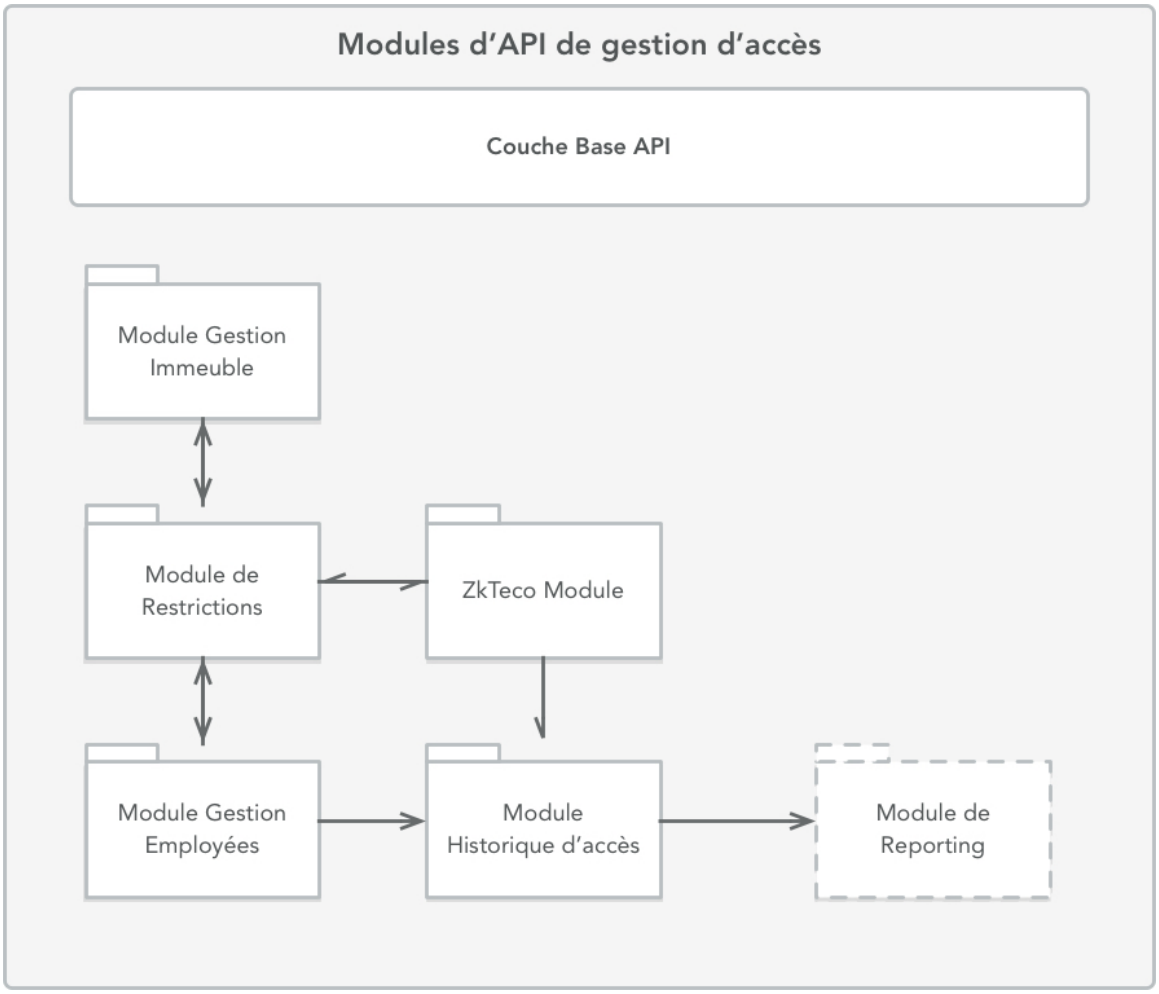

Figure 24 ***Diagramme des modules d'API de gestion d'accès***

Cet diagramme de module représente la disposition du système de gestion d'accès. Chaque module est une extension des module de la couche de base. On va développer sur pour chaque module les classe qui le composent.

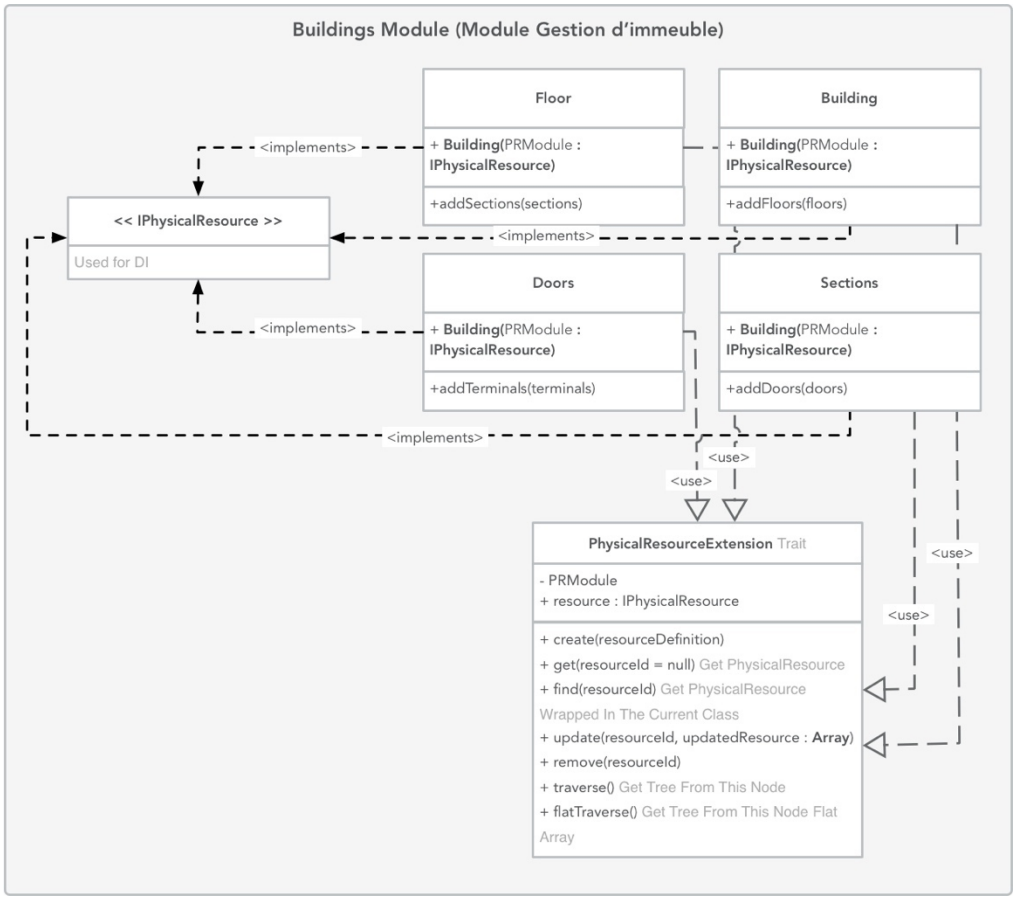

Figure 25 ***Diagramme de class du module de gestion d'immeuble***

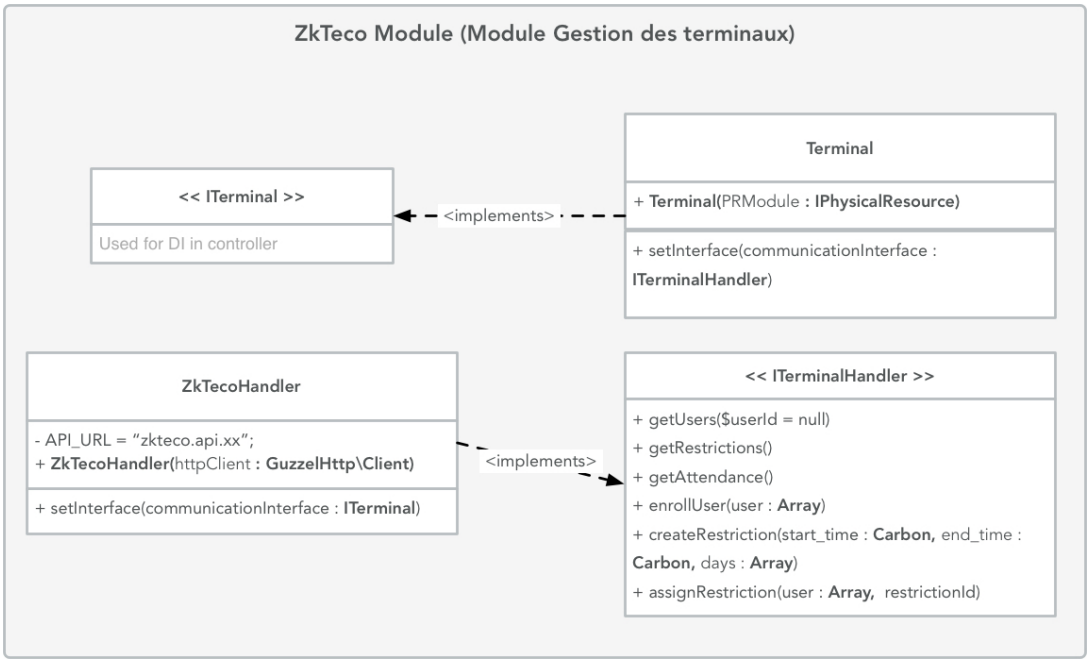

Figure 26 ***Diagramme de class pour le module asynchrone de gestion des terminaux***

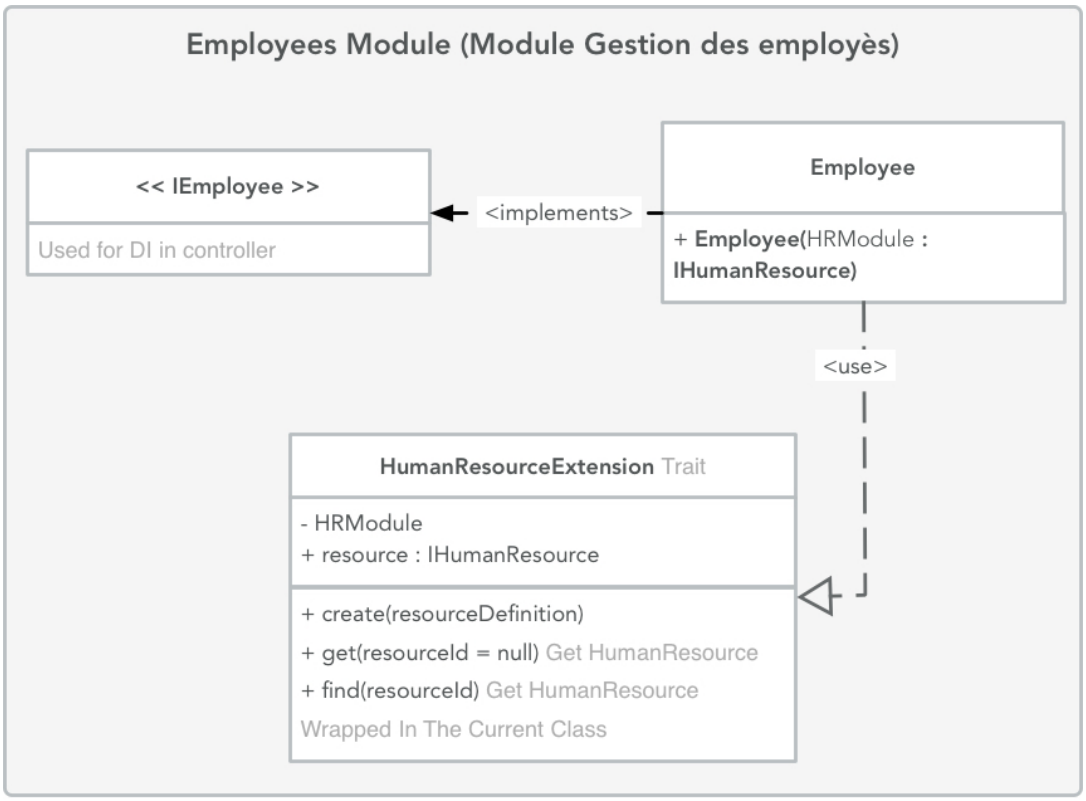

Figure 27 ***Digramme de class pour module d'employés***

```php
<?php

namespace App\Access\Buildings;

use App\Access\Buildings\Floor;
use App\Access\Traits\PhysicalResourceExtension;
use App\Base\Resources\Physical\Contracts\IPhysicalResource;



class Building implements IPhysicalResource
{
  use PhysicalResourceExtension;
  public $floors;
  public function __construct(IPhysicalResource $PRModule)
  {
    $this->PRModule = $PRModule;
  }

  /**
   * Add Floors
   */
  public function addFloors($floors)
  {
    if ($floors instanceof Floor) {
      $this->resource->appendNode($floors->resource);
    } else {
      foreach ($floors as $floor) {
        $this->resource->appendNode($floor->resource);
      }
    }
    $this->floors = $this->resource->children()->get();
  }

}
```

Figure 28 ***Classe de Building étendu du module de ressource physique***

Comme on voit dans la *figure 29* l'injection du module de ressource physique depuis son interface. Cela donnera toutes ces fonctionnalités à cette class sans qu'elles soient liées avec un héritage simple (Class Parent – Class Fils). De plus cela rendra nos extensions plus testables et avec moins de risques de point de défaillance unique. La trait **PhysicalResourceExtension** contient les méthodes qui aident à manipuler les ressources

physiques, tel que les opérations CRUD[27] et d'autres interactions qu'une ressource physique peut avoir. Il y a un trait similaire pour les extensions des ressources humaines.

- **Front-end Web Prototypage**

  Pour créer le front-end, nous avons traversé une phase de prototypage papier, pour mieux visualiser comment l'interface utilisateur va être organisé avant de l'implémenter avec **React**. Puisque ce Framework utilise une architecture basée sur les composants, ce prototype de papier nous a vraiment aidé à définir nos composants avant de les programmer.

[27] Create Read Update Delete

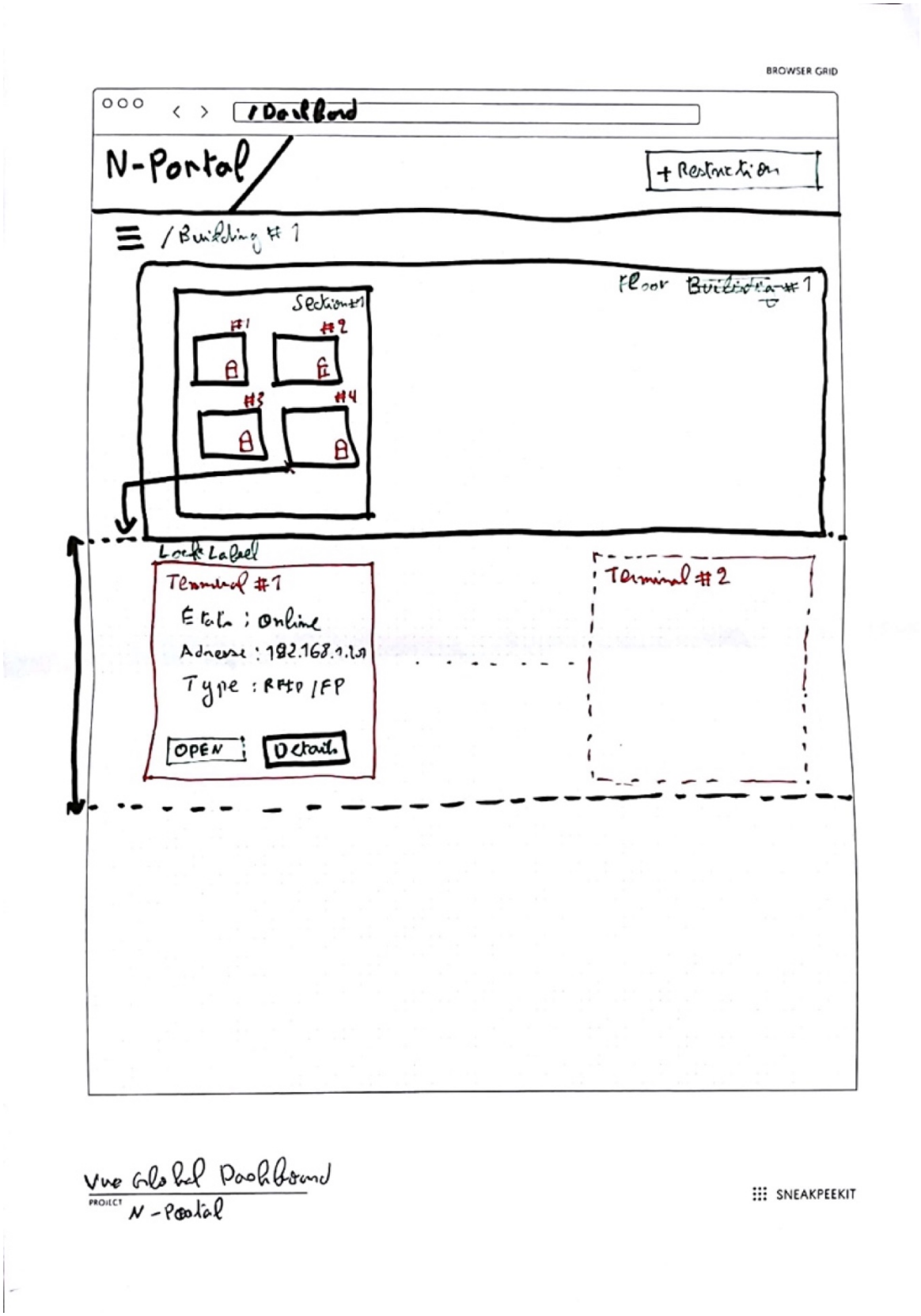

Figure 29 ***Vue Dashboard (Prototypage Rapide)***

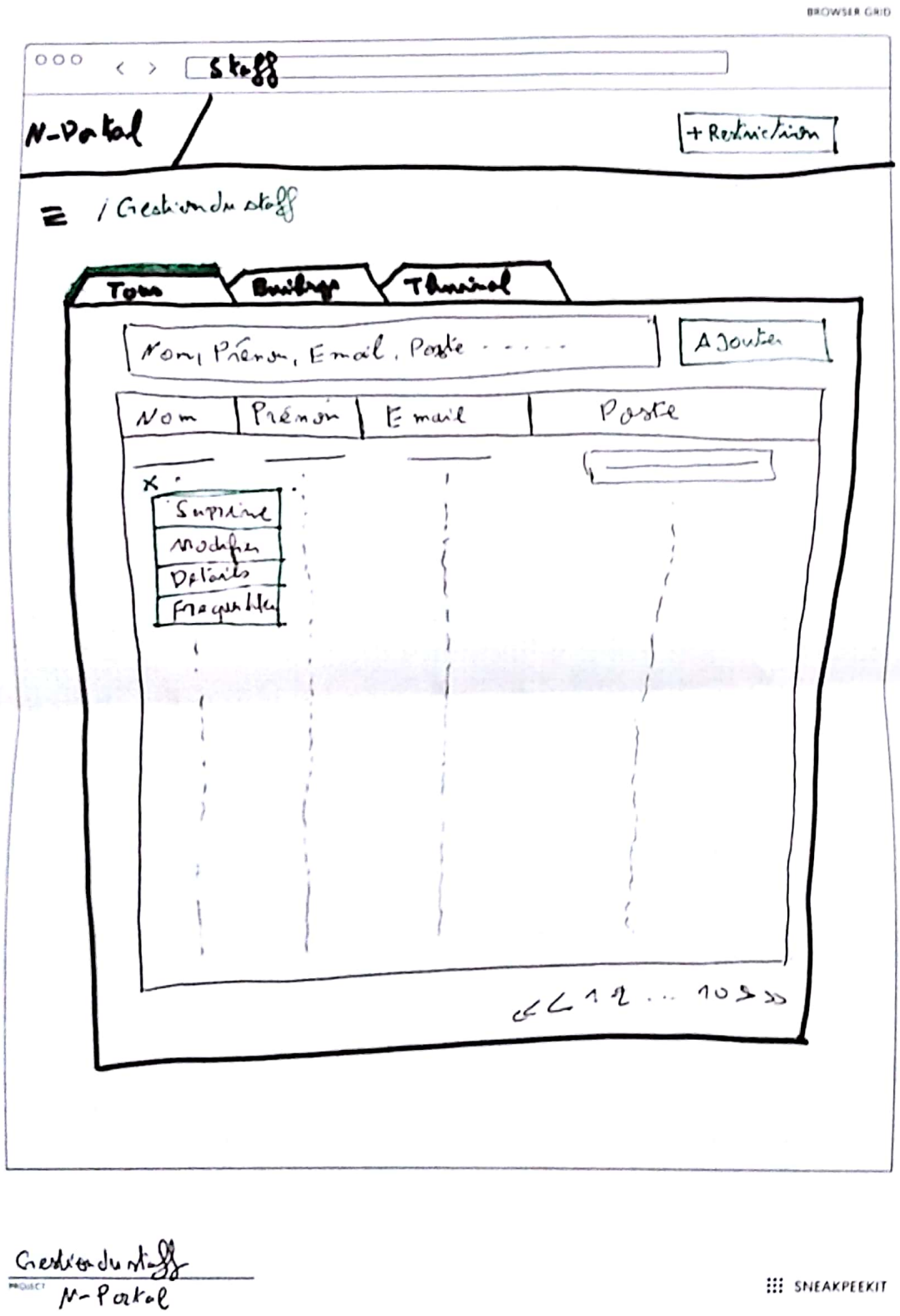

Figure 30 ***Vue Employés (Prototypage Rapide)***

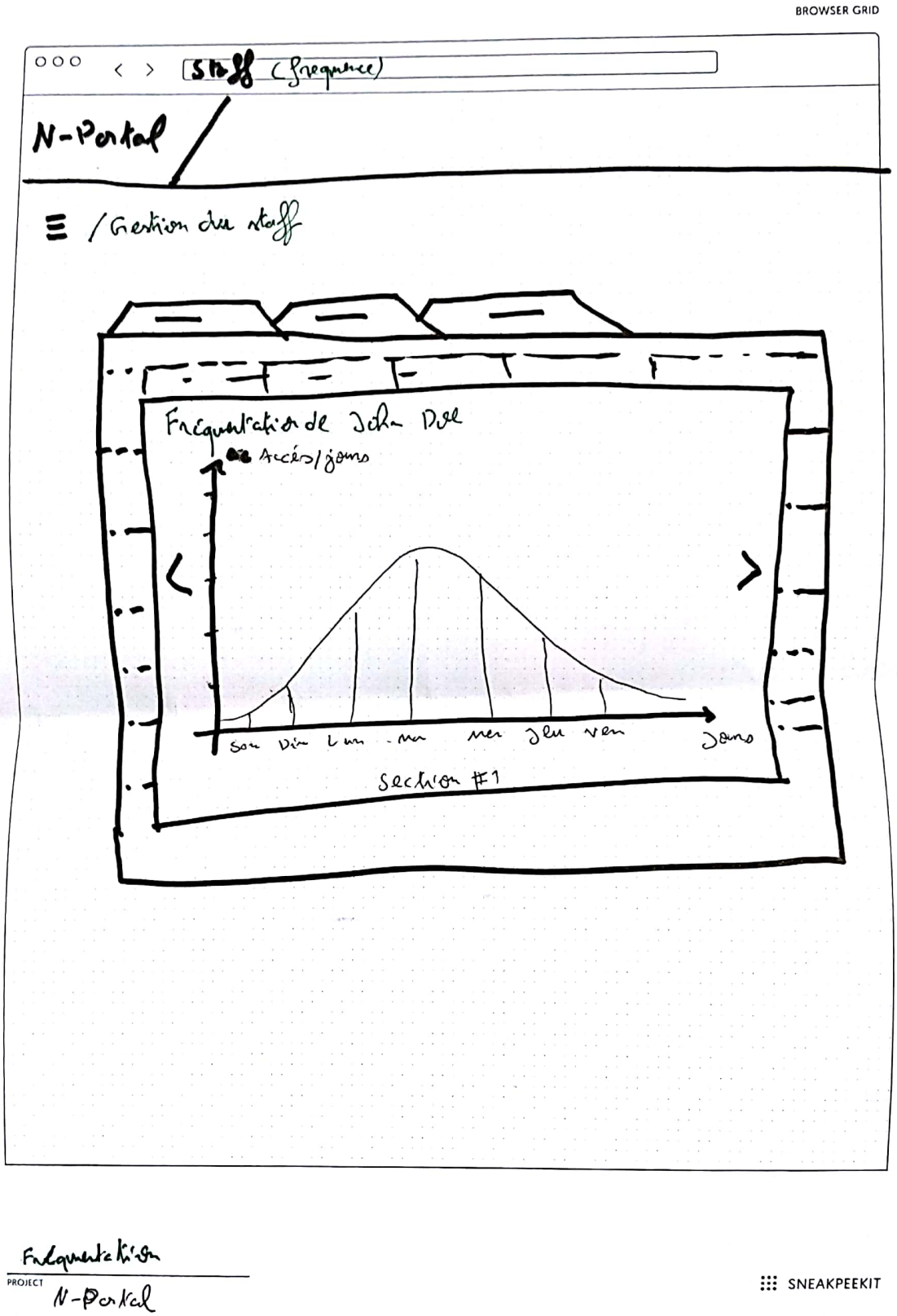

Figure 31 ***Vue de fréquentation d'employé (Prototypage Rapide)***

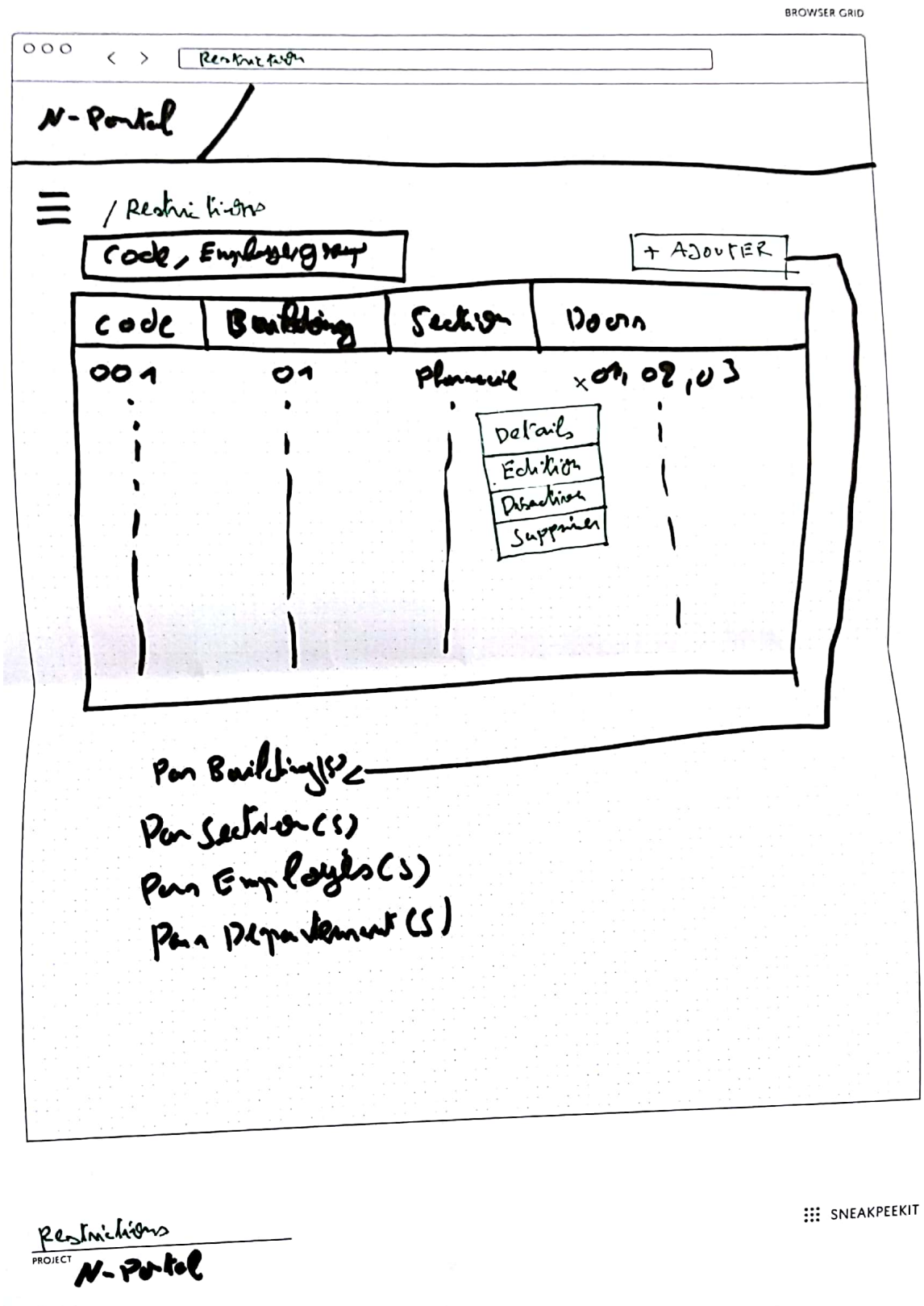

Figure 32 ***Vue de restrictions (Prototypage Rapide)***

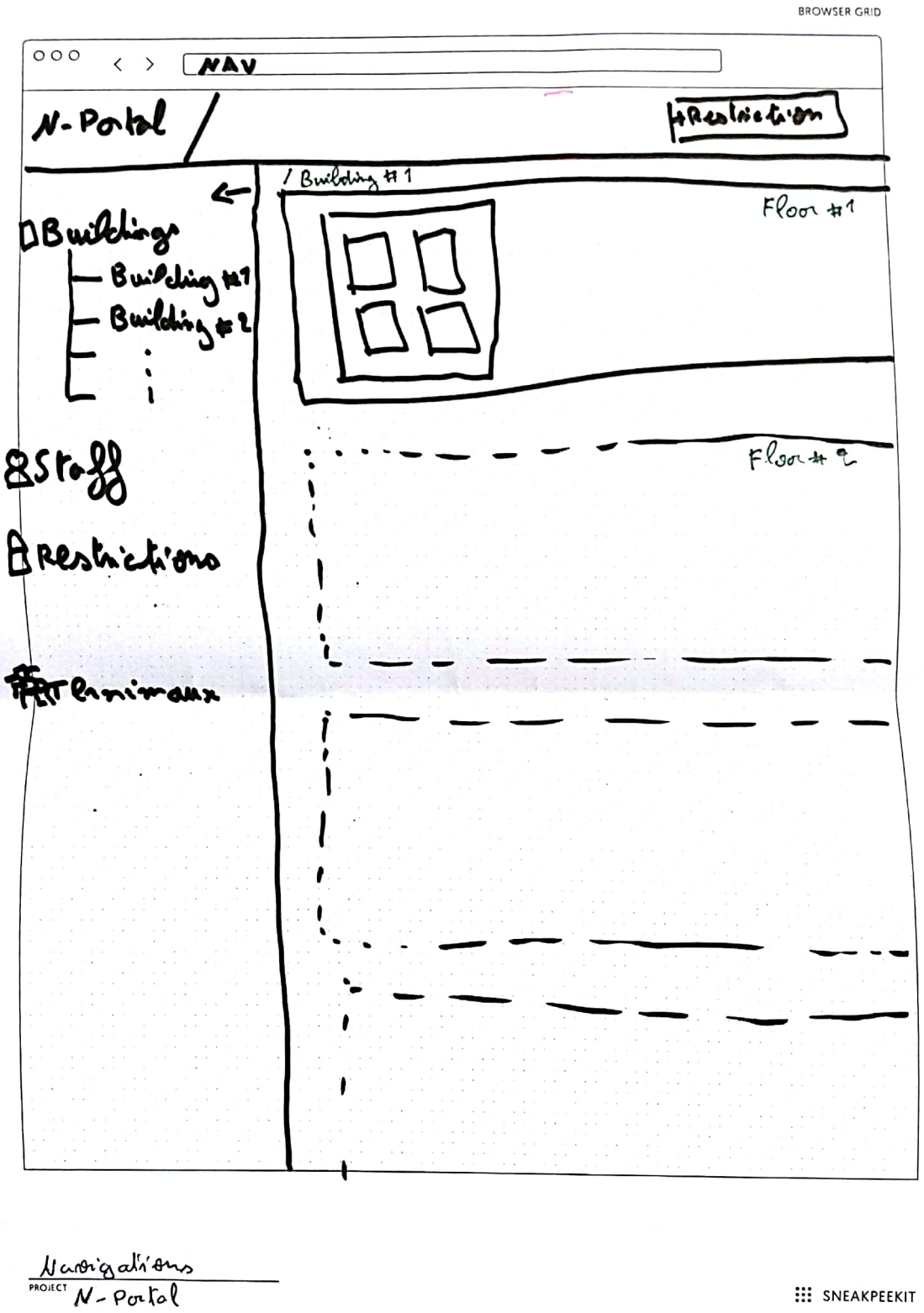

Figure 33 ***Vue dashboard avec menu sidebar (Prototypage Rapide)***

- **Front-end Implémentation**
  - **ReactJs :**

    Un Framework développé par **Facebook** pour le développement des interface web et mobile, sa particularité est son implémentation du rendu des éléments de la page web. Ou il génère une version virtuel du document de la page HTML appelé **Virtual-Dom**[28] et fais une comparaison avec le vrais **DOM**[29] puis il va seulement mettre à jour l'élément qui a changé. Cella le rend très optimisé pour la création des interfaces web interactives.

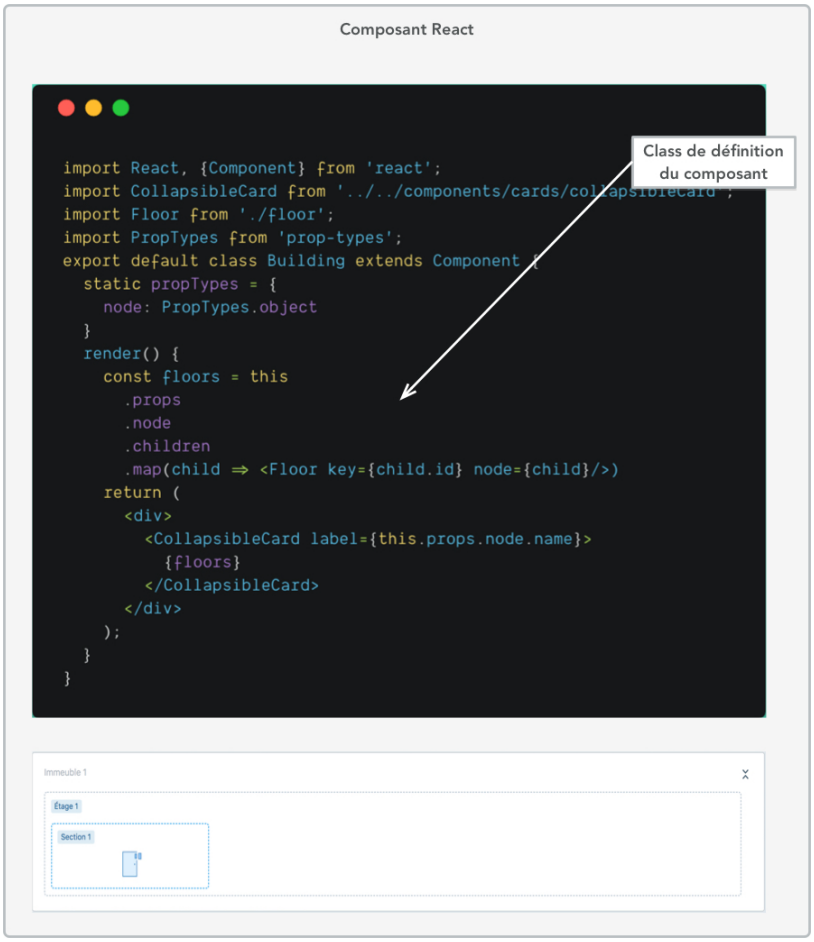

Figure 34 ***Définition de composant Reactjs***

[28] Représentation en objet de l'arbre DOM
[29] Document Object Model, arbre qui représente le document HTML (Hypertext Markup Language)

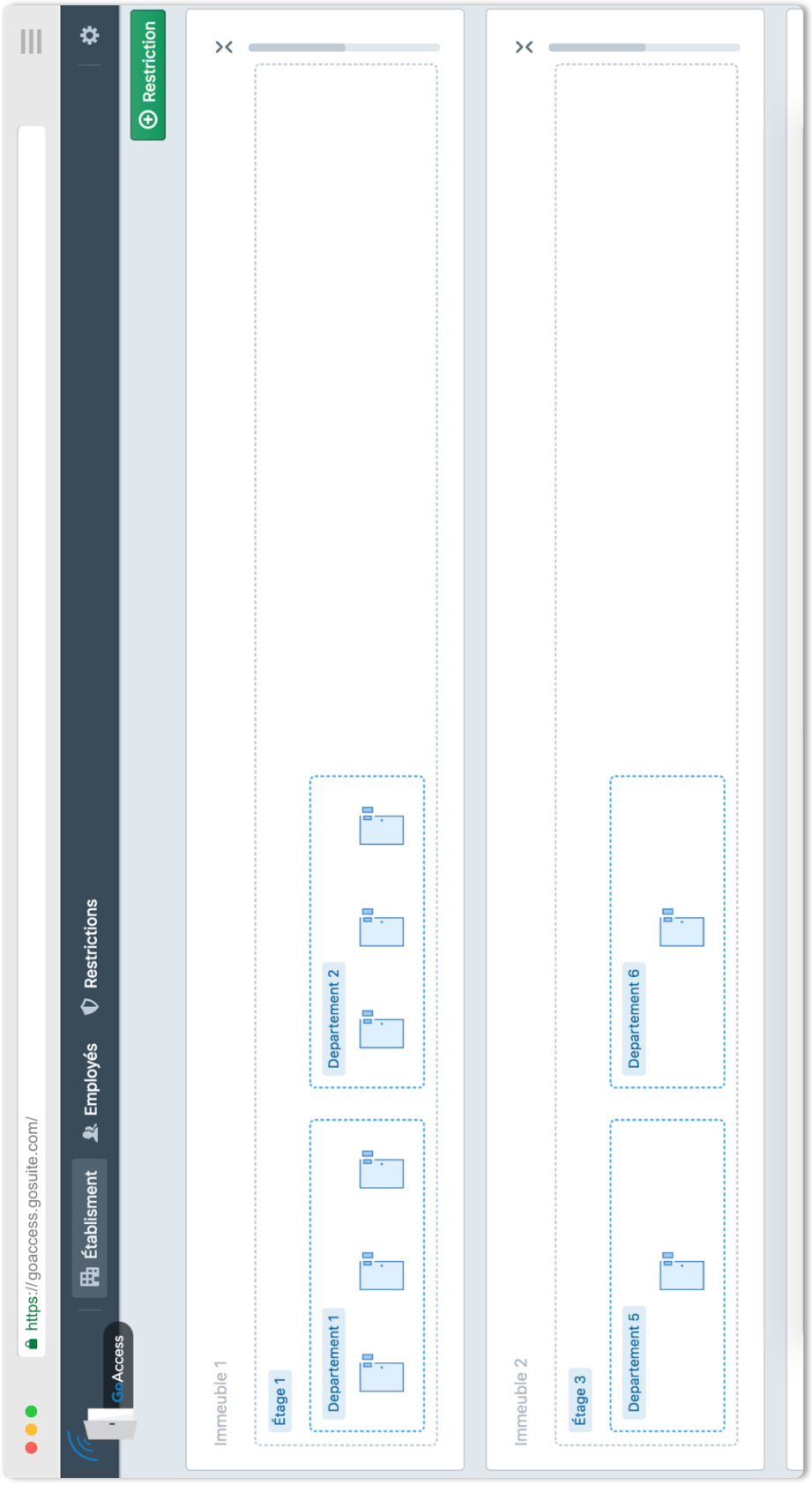

Figure 35 ***Dashboard donne une vue général de l'établissement***

Vue principale qui démontre la structure de l' organisation, et les portes connectées.

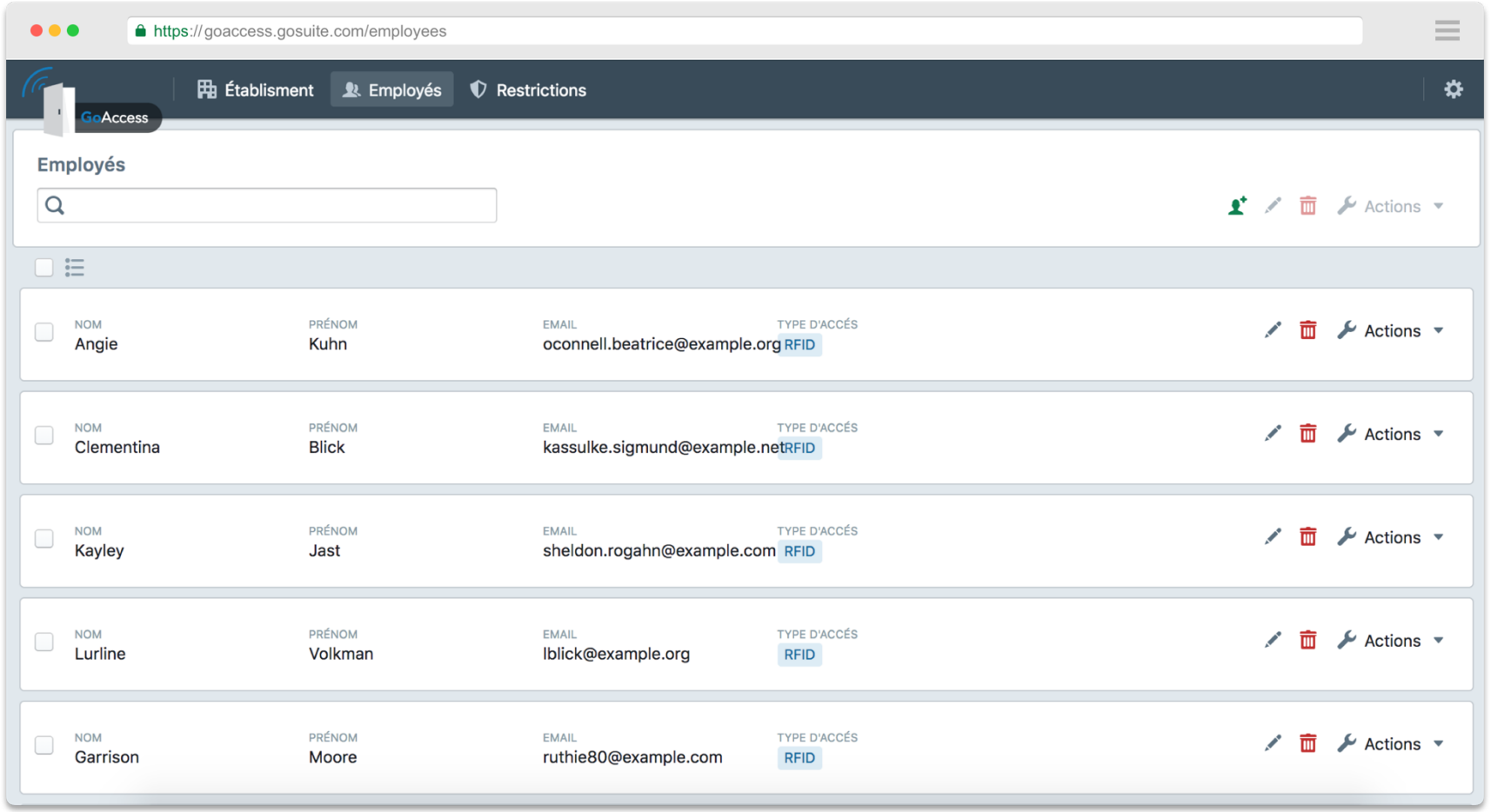

Figure 36 ***Listing des employées***

Vue des employés qui démontre leur liste avec des actions de manipulation.

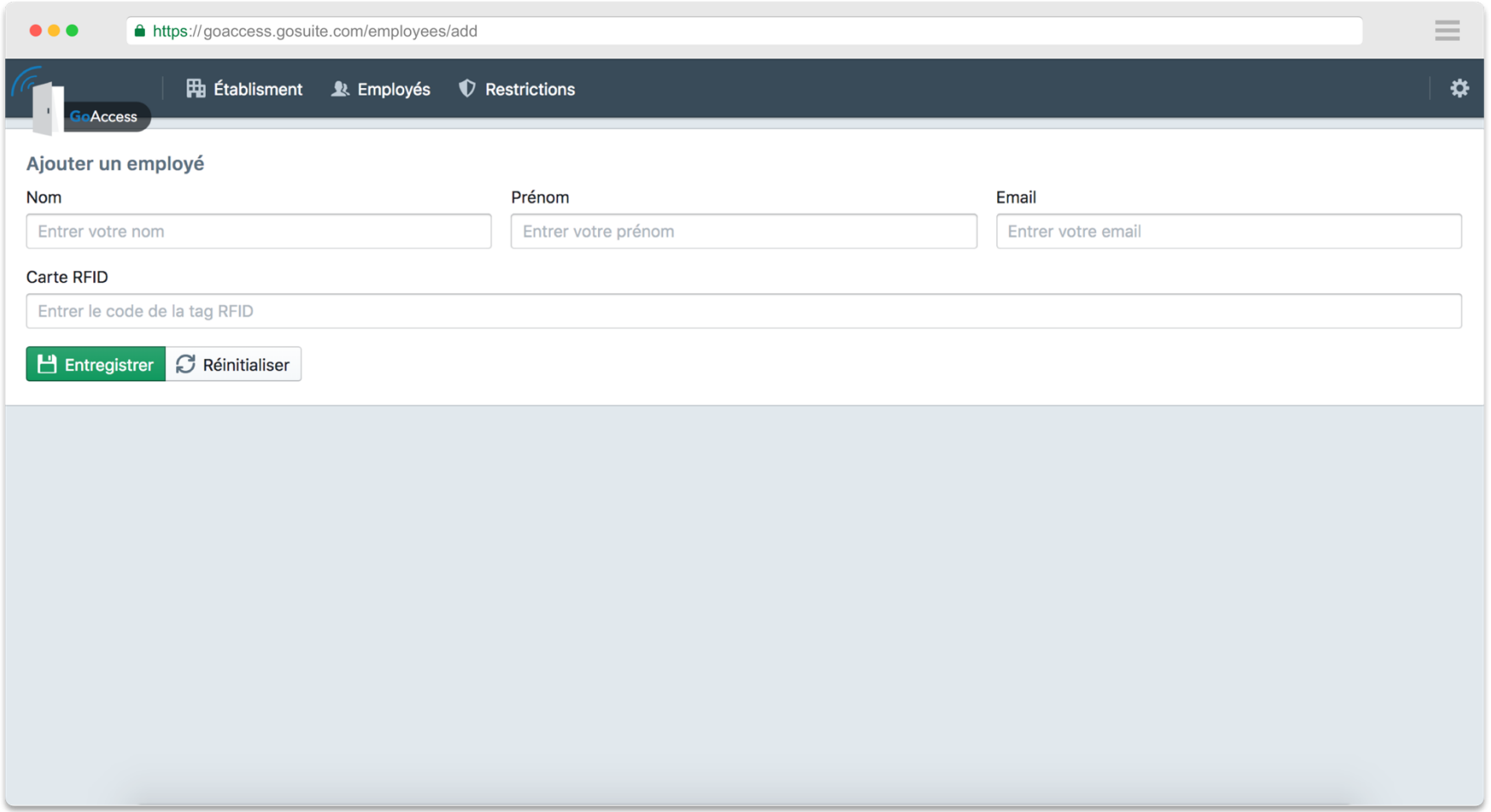

Figure 37 ***Formulaire d'ajout d'employés***

Formulaire d' ajout d' employé dans la base de donnée des systèmes de gestion d' accès.

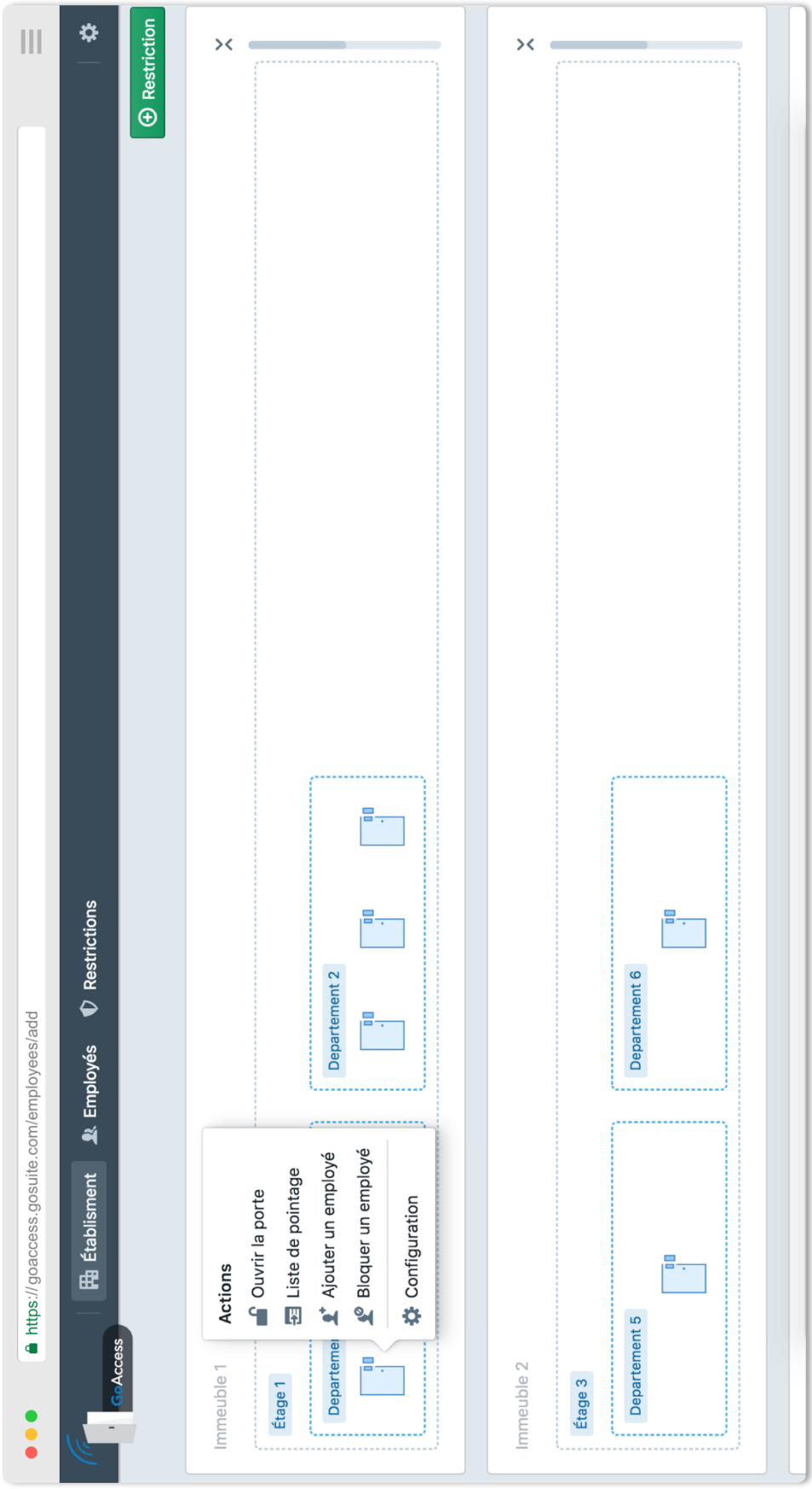

Figure 38 ***Actions rapides sur les portes***

La liste des actions sur les portes pour les interaction avec les terminaux de serrure.

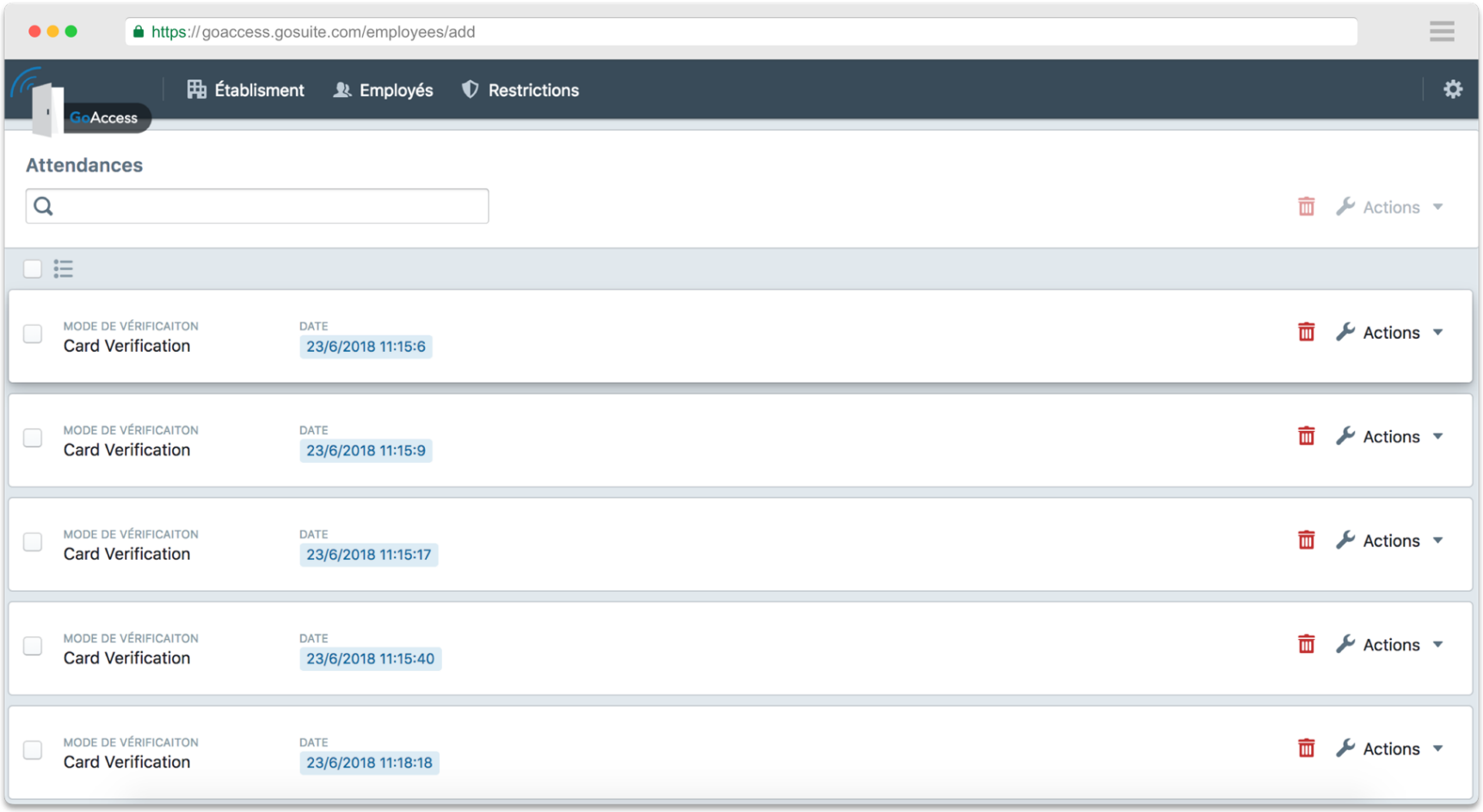

Figure 39 ***Listing du pointage de la porte sélectionnée***

Liste de pointage par porte, avec les modes de vérification des utilisateurs.

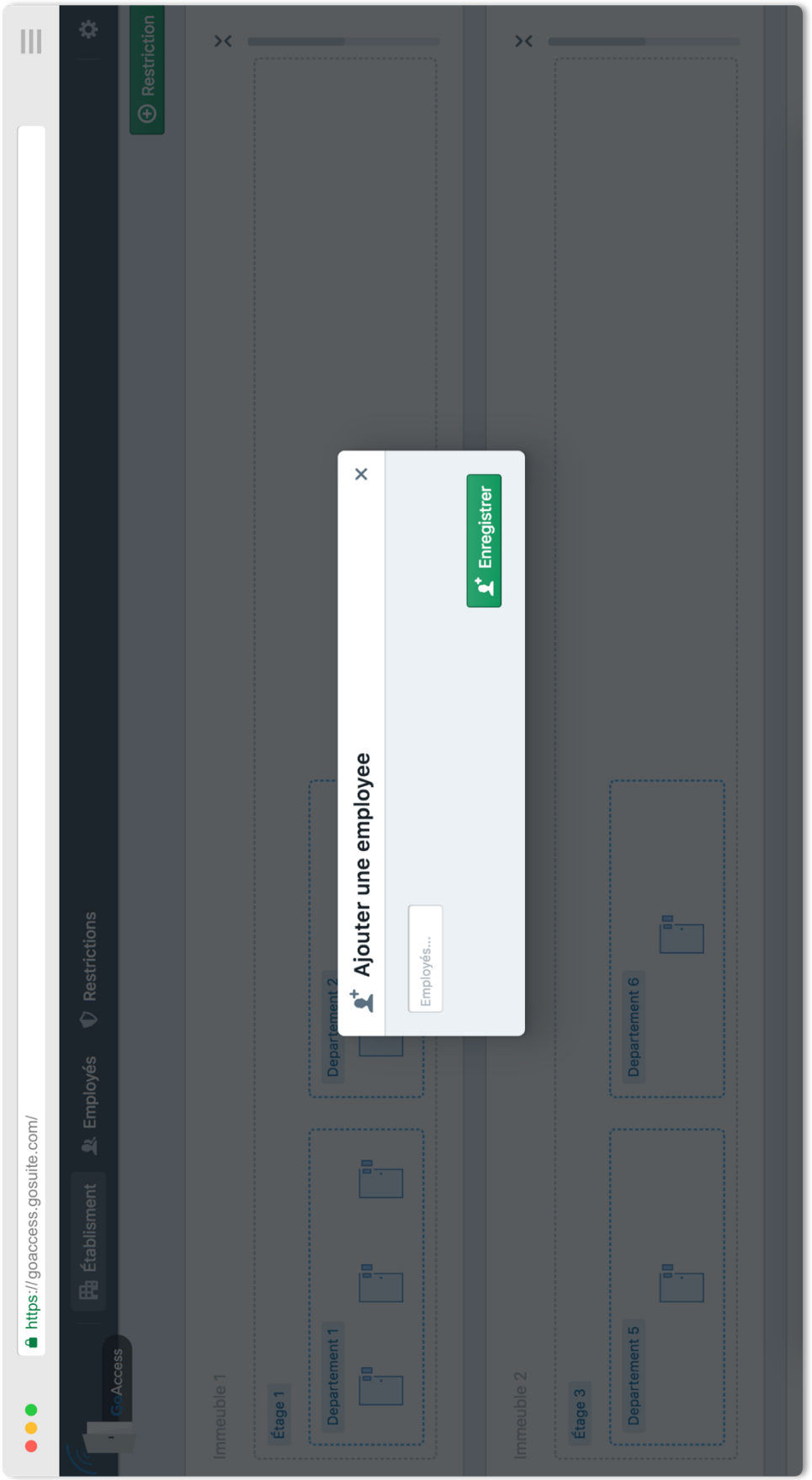

Figure 40 ***Ajout d'employé dans une porte***

Ajout d' employés pour l' accès dans une porte.

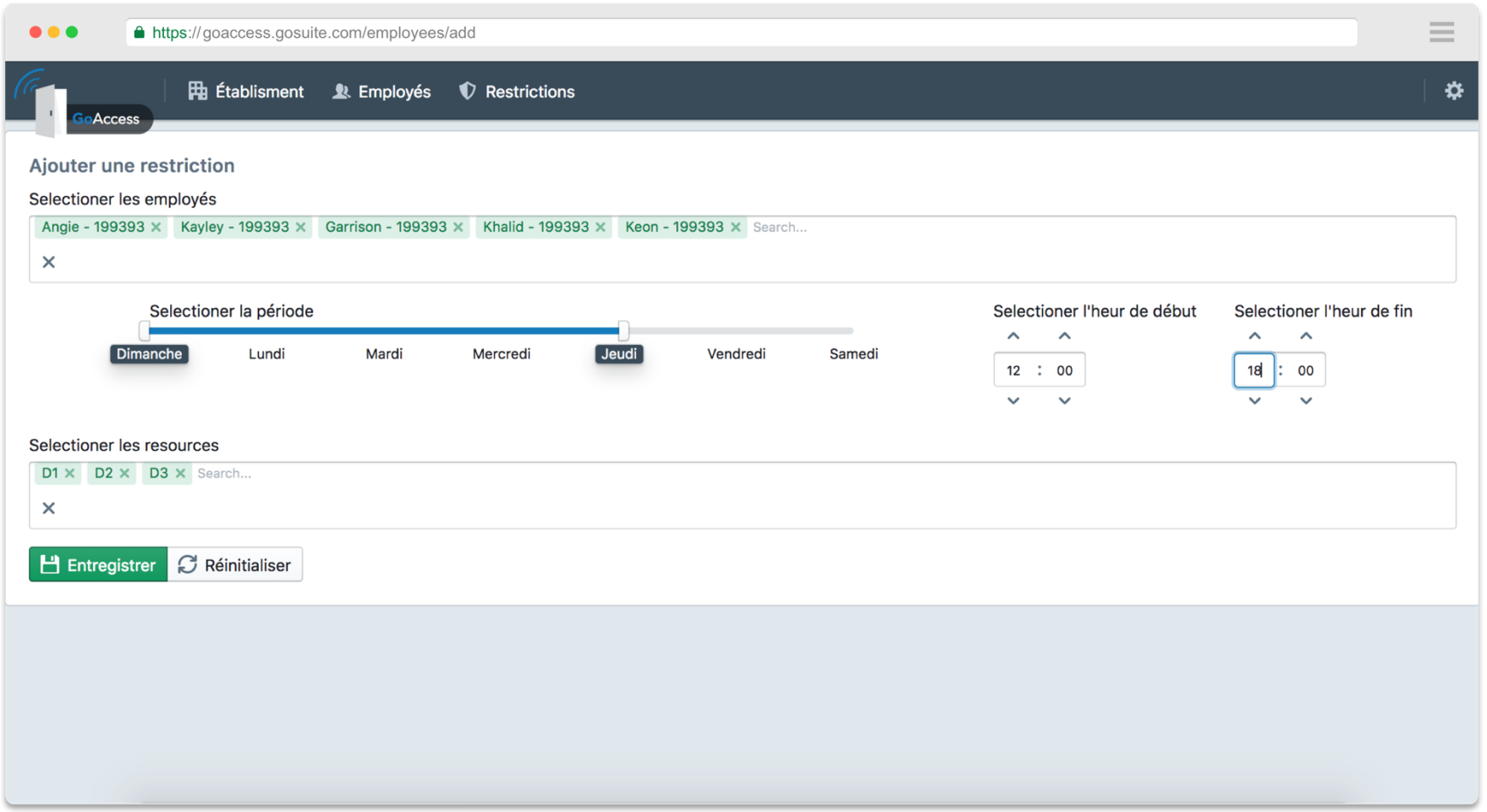

Figure 41 ***Formulaire de restrictions par période***

L'ajout de restriction pour les employés dans une période pour les ressources sélectionnées.

5.7 Étape 7 : Analyser le design actuel et revoir l'objectif de l'itération et son achèvement

Dans cette itération on a adressé les cas d'utilisation pour arriver à un MVP[30]

Non Traité	Traité Partialement	Complètement Traité	Choix et décision de conception
UC-7-A	**UC-1-A**		On a créé les extensions nécessaires depuis le module de ressource physique, pour avoir un module de gestion d'institution ces immeubles, étages, sections, portes et terminaux associés aux portes. Dans cette itération on ne supporte que la lecture de ces éléments dans le front-end.
UC-8-A	**UC-2-A, UC-3-A, UC-4-A, UC-6-A**		On a créé un **API RPC** en ASP.NET qui communique avec les terminaux **ZkTeco** et qui intègre la quasi-totalité des fonctionnalités du terminal d'accès. Puis on a créé un module interne qui fait partie de l'API de gestion d'accès : le module **ZkTeco**. Ce module est un client http qui prend les requêtes destinées aux terminaux qui vienne vers l'API de gestion d'accès et les envoie vers l'API RPC pour commander le terminal. d'où on implémente l'ajout des cartes **RFID**, **liste de pointage** et **restrictions**

Tableau 22 Étude de cas system de gestion d'accès ***- Système de gestion d'accès – Analyse du design actuel***

[30] Minimum Viable Product

6. Outils utilisé dans le flux de développement

Comme nous l'avons mentionné précédemment, notre workflow de développement utilise **TDD**, mais cela nous oblige à automatiser le processus d'intégration, pour mettre en œuvre un tel flux de travail, nous utilisons **Git** comme un système de contrôle de version avec une méthodologie **GitFlow**. Ensuite, nous déployons notre code source sur **Gitlab** où nous implémentons notre serveur d'intégration Continu via Docker, ceci nous permet de déployer l'application en exécutant nos suites de tests et de voir si elles réussissent seulement dans ce cas la branche déployée sera fusionnée dans la branche master.

- **Git**

  Def Wikipedia :

  git est un logiciel de gestion de versions décentralisé. C'est un logiciel libre créé par **Linus Torvalds**, auteur du noyau Linux, et distribué selon les termes de la licence publique générale **GNU version 2**. En **2016**, il s'agit du logiciel de gestion de versions le plus populaire qui est utilisé par plus de **douze millions de personnes**. [37]

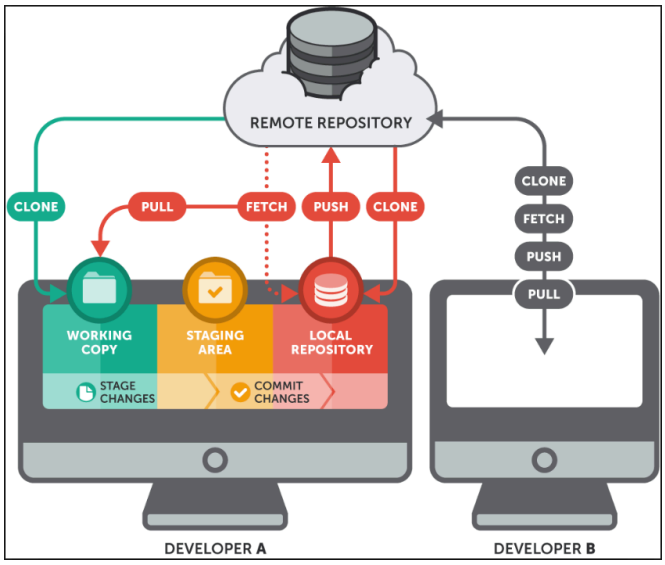

Figure 42 ***Architecture distribué du git***

Détails :

Outils de gestion de source code collaboratif d'une façon distribué, ou chaque développeur a son propre code source dans sa machine (**Working Directory**), qui sera synchronisé avec les autres développeurs dans un serveur centralisé (**Remote Repository**). En théorie Git est un système qui journalise les modification du développeur. Chaque modification (Ajout, Edition, Suppression) doit être marquée par un **commit** chaque commit est une version du projet dans un temps donné. Les commits sont groupés dans des **branches** par default tous projet a une **branche** principale appelé **Master**.

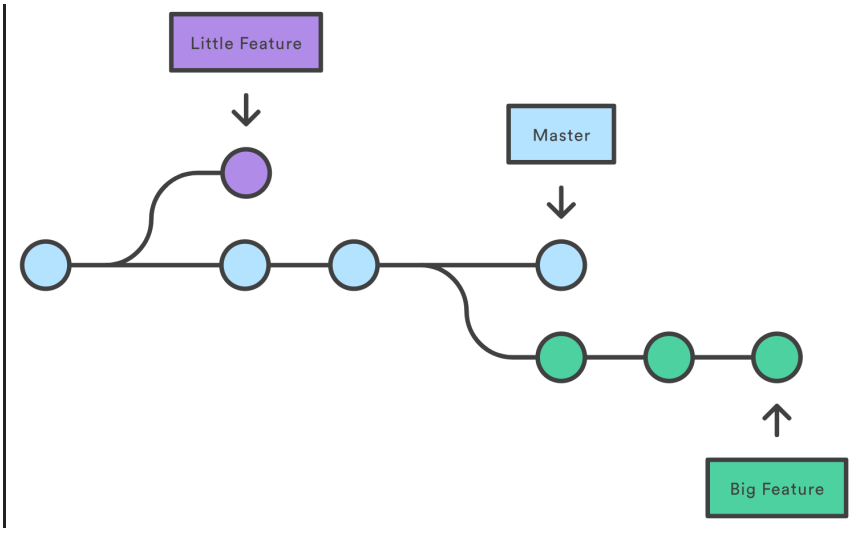

Figure 43 ***Les branches et commits dans un arbre git***

- **GitFlow**

C'est une méthode instauré par **Vincent Driessende** pour utiliser git d'une façon plus organisé et expressive qui favorise la séparation des responsabilités. Ou il propose une stratégie de deviser les branches en **Features (Fonctionnalité), Releases (Distribution Client), Develop (Branch de développement)** et enfin une branche optionnelle **Hotfix (Rectifications Urgentes)**

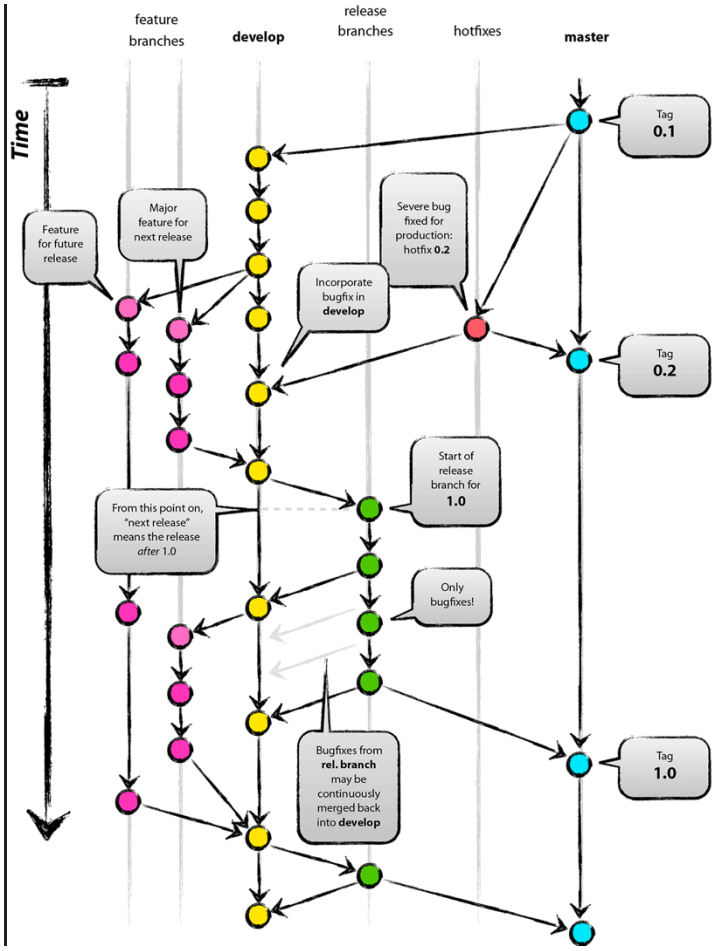

Figure 44 ***Gitflow Workflow***

Cette organisation nous permet entre développeur de savoir l'évolution des fonctionnalités et les distribution. De plus pour un meilleur suivi il est conseillé d'utilisé un outil graphique tel que **GitKraken**.

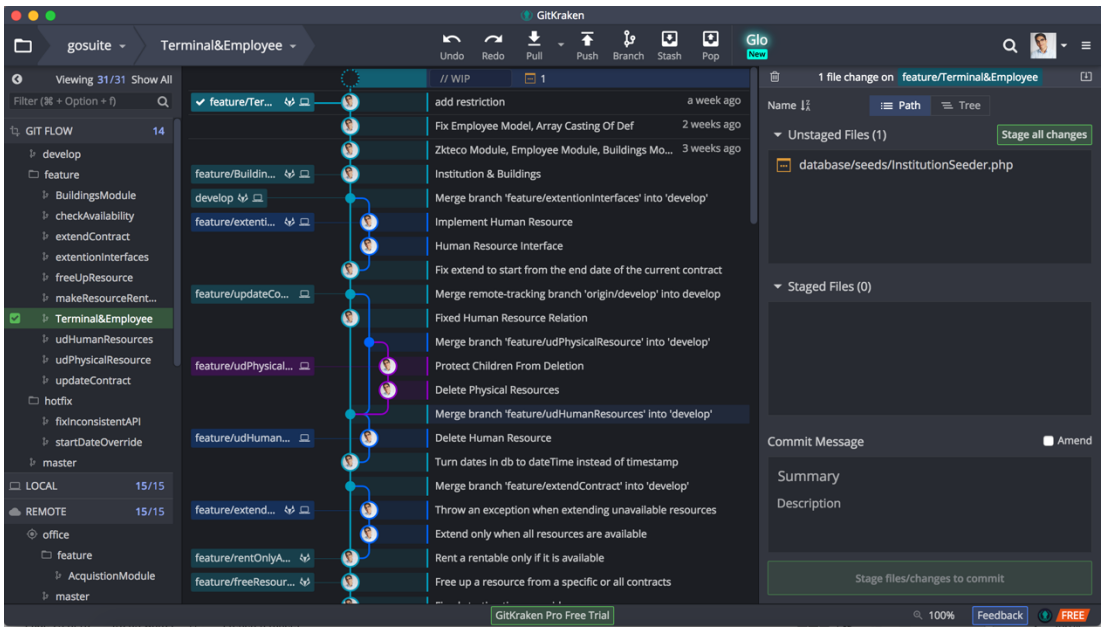

Figure 45 ***GitKraken Projet Gosuite (Étude de Cas)***

- **GitLab**

Platform **PAAS** pour gestion de code source avec Git, offre un service d'intégration continue support déploiement **Docker** pour intégration / distribution continue. De plus il offre une gestion de projet **Kanban.** On l'a utiliser pour les trois niveau de notre système ; **Base API / Acces API** , **Front-end** et **Zkteco RPC API**

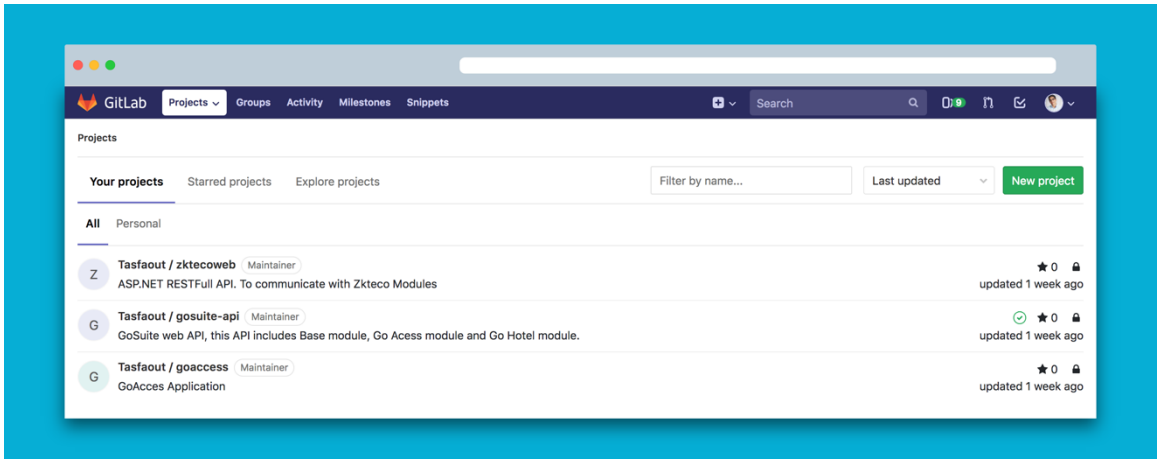

Figure 46 ***Gitlab Sources***

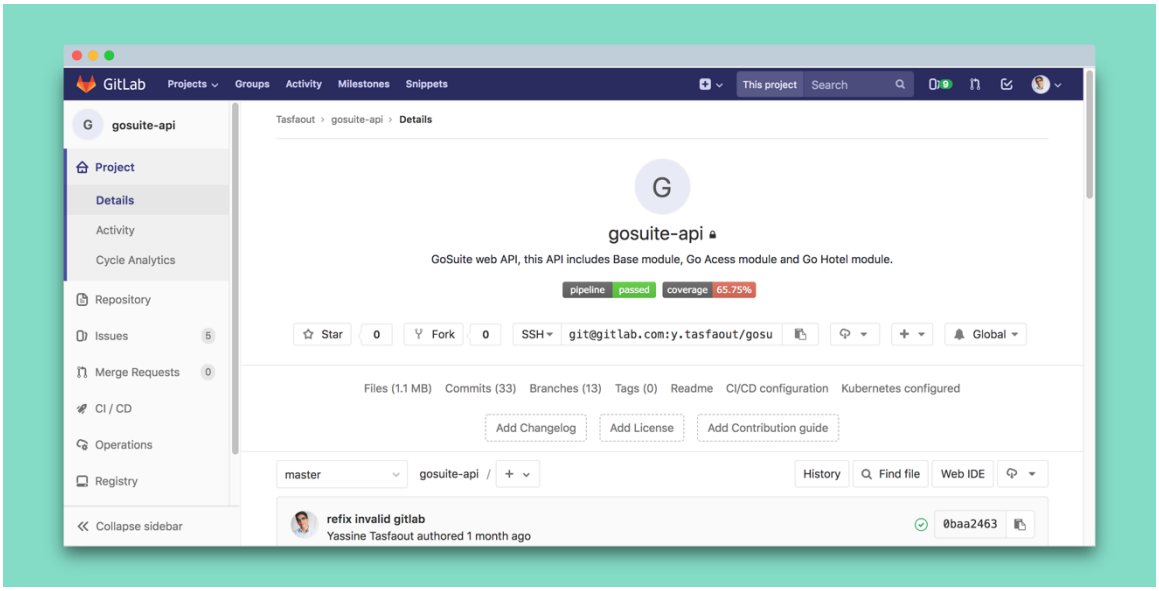

Figure 47 ***Platform de gestion Gitlab du projet avec les badges des tests***

On voit à partir du badge pipeline passed que les tests unitaires sont validés. Et l'application est prêt pour être déployé en production.

- **Docker**

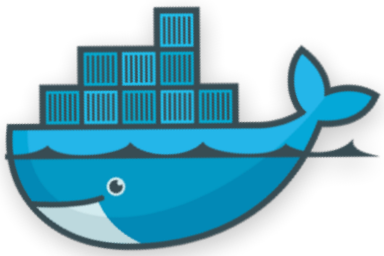

Figure 48 ***Logo Docker***

Docker est un outils récent de déploiement, basé sur la ligne de commande, il créer un environnement d'exécution en utilisant une technologie **UNIX** des **Centenaires**. Les centenaires sont une bonne pratique de déploiement utilisée depuis longtemps, mais leur instanciation manuel était toujours difficile. Un

centenaire mal configuré peut être risque de sécurité sur la machine. D'où viens la solution Docker, avec docker on peut créer des centenaires facilement car l'engin de docker s'occupe du reste de la configuration. Il ne faut pas confondre un centenaire avec une machine virtuelle, un centenaire est un environnement d'exécution clos isolé qui se pose sur un Kernel Linux. Par default un centenaire n'a aucun accès direct avec le Kernel seulement si explicitement définit. Contrairement à une machine virtuelle ou il y a une allocation de ressource matérielle pour tout un système d'opération, ceci rend les centenaires très performant. Dans la figure qui suit, un exemple de configuration docker avec **docker compose**[31]

[31] Utilitaires d'orchestration entres les centenaires

```yaml
nginx:
  build: ./nginx
  container_name: nginx-container
  restart: always
  command: nginx -g "daemon off;"
  links:
    - php
    - realtime
  ports:
    - "3001:80"
php:
  build: ./php
  container_name: php-container
  working_dir: /var/www/html/public
  command: php-fpm
  volumes:
      - /Users/yassine/Projects/gosuite/:/var/www/html/
  restart: always
  links:
    - mysql
    - cache
  ports:
    - "9000"
  environment:
    APP_ENV: local
    APP_DEBUG: 'true'
    APP_KEY: Z29ob3RlbF9yZXN1cmVjdGlvbg==
    APP_LOCALE: fr
    APP_FALLBACK_LOCALE: en
    APP_TIMEZONE: Africa/Algiers
    DB_CONNECTION: mysql
    DB_HOST: mysql
    DB_DATABASE: gosuite
    DB_USERNAME: gosuite_root
    DB_PASSWORD: root
    CACHE_DRIVER: redis
    BROADCAST_DRIVER: redis
    QUEUE_DRIVER: sync
```

Figure 49 ***Exemple de configuration Docker***

- **Visual Studio Code**

Un éditeur de code puissant de **Microsoft**, créer par les développeur de **Eclipse IDE**[32]**.** Cet outil est gratuit open source et à des tonne d'extension pour aider l'environnent de développement, et il support un très bon **autocomplete** du code mieux que les autres éditeurs gratuits.

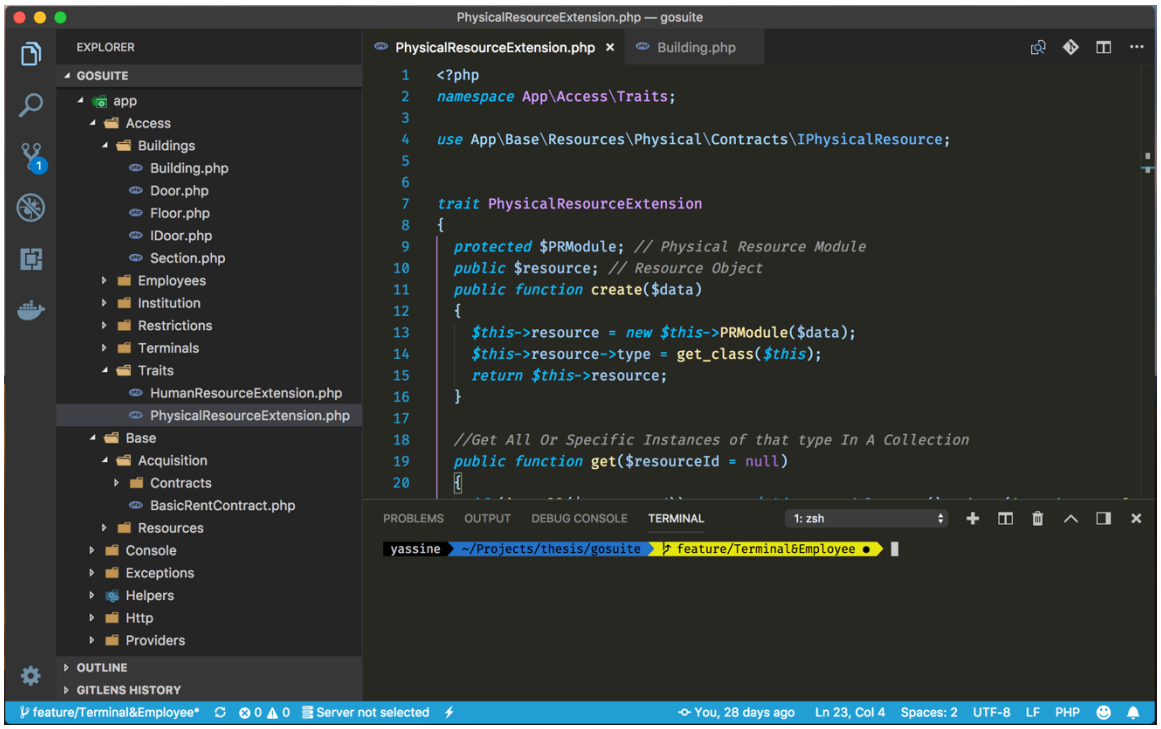

Figure 50 ***Visual Studio Code - Microsoft***

[32] Integrated Development Environment

7. Conclusion

Dans ce chapitre final, on a vu comment déployer ADD pour un système Greenfield. De plus, on a pu voir comment on passe des cas d'utilisations, attributs de qualité, contraintes et soucis architecturaux aux éléments du System Design. ADD est un modèle simple à implémenter et facilite la documentation de la conception du système car il souligne davantage la justification des décisions dans chacune de ses étapes. Puis sa nature itérative le rend parfait pour les environnements agiles. Et enfin on a parlé des outils utilisé dans notre flux de travail.

Conclusion générale et Perspectives

Au long de ce travail, nous avons commencé une feuille de route pour la conception d'un système évolutif avec pour objectif principal d'être extensible. Et permettre à son tour la création des sous-systèmes orientés vers la gestion d'entreprise. Nous avons exploré les concepts de l'architecture des systèmes et élaboré sur la méthode ADD. Cependant, un modèle d'architecture seul n'est pas suffisant pour construire un système. Pour cela, nous avons introduit les concepts d'architecture pour élargir nos connaissances et gagner davantage en référence lors de la conception des systèmes pour résoudre les problèmes récurrents. Ensuite, nous avons complété cela avec la gestion des projets software et les types d'environnement ; Waterfall et Agile. Nous avons également discuté des meilleures pratiques de développement et de programmation orientée-objet. Enfin, nous avons rassemblé toutes ces connaissances acquises et les avons mises en pratique dans notre étude de cas où nous avons rapidement établi notre architecture initiale en satisfaisant nos attributs de qualité et nos exigences fonctionnelles définies.

Avec la collaboration de **NetSprint**, nous voulons transformer ce système en un Framework de base pour les systèmes d'information dans le domaine de gestion interne et externe des entreprises ; hôtellerie, commercialisation, stock, laboratoires … etc. En ce qui concerne le système de gestion d'accès, l'étape suivante sera de passer à d'autres itérations dans le processus ADD pour satisfaire les cas d'utilisation, attributs de qualités restantes. De plus, voir d'autre exigences potentielles et les planifier dans des itérations futures, afin d'avoir un système implémenté et prêt à être distribué. En suivant la méthodologie TDD dans la programmation et un modèle d'architecture convenable tel que ADD dans une équipe agile qui utilise Kanban, on pourra accomplir l'objectif de livrer des produits de haute qualité et dans les attentes des clients.

Références Bibliographiques

[1] H. Cervantes et R. Kazman, *Designing Software Architectures_ A Practical Approach.pdf*. 2016.

[2] W. G. Wood, « A Practical Example of Applying Attribute-Driven Design (ADD), Version 2 . 0 », *Technology*, vol. Version 2, nº February, p. 59, 2007.

[3] R. Wojcik et P. Clements, « Attribute-Driven Design (ADD), Version 2 . 0 », nº November, 2006.

[4] Paper Rational Software White, « Rational Unified Process Best Practices for Software », *Development*, p. 1-21, 2004.

[5] D. Ström, « Purposes of Software Architecture Design », nº January, 2005.

[6] M. O. Ahmad, J. Markkula, et M. Oivo, « Kanban in software development: A systematic literature review », *2013 39th Euromicro Conf. Softw. Eng. Adv. Appl.*, p. 9-16, 2013.

[7] M. Senapathi, P. Middleton, et G. Evans, « Factors affecting effectiveness of agile usage - Insights from the BBC worldwide case study », in *Lecture Notes in Business Information Processing*, 2011, vol. 77 LNBIP, p. 132-145.

[8] V. Gulliksen Stray, N. B. Moe, et T. Dingsøyr, « Challenges to teamwork: A multiple case study of two agile teams », in *Lecture Notes in Business Information Processing*, 2011, vol. 77 LNBIP, p. 146-161.

[9] M. Ikonen, E. Pirinen, F. Fagerholm, P. Kettunen, et P. Abrahamsson, « On the impact of Kanban on software project work: An empirical case study investigation », in *Proceedings - 2011 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011*, 2011, p. 305-314.

[10] R. Polk, « Agile & Kanban in coordination », in *Proceedings - 2011 Agile Conference, Agile 2011*, 2011, p. 263-268.

[11] K. Rutherford, P. Shannon, C. Judson, et N. Kidd, « From chaos to kanban, via Scrum », in *Lecture Notes in Business Information Processing*, 2010, vol. 48 LNBIP, p. 344-352.

[12] J. O. Birkeland, « From a timebox tangle to a more flexible flow », in *Lecture Notes in Business Information Processing*, 2010, vol. 48 LNBIP, p. 325-334.

[13] T. Wijewardena, « Do you dare to ask your HR manager to practice KANBAN?: The experience report of an offshore software company in Sri Lanka introducing agile practices into its Human Resource (HR) department », in *Proceedings - 2011 Agile Conference, Agile 2011*, 2011, p. 161-167.

[14] K. Greaves, « Taming the customer support queue: A Kanban experience report », in *Proceedings - 2011 Agile Conference, Agile 2011*, 2011, p. 154-160.

[15] M. Taipale, « Huitale - A story of a Finnish lean startup », in *Lecture Notes in Business Information Processing*, 2010, vol. 65 LNBIP, p. 111-114.

[16] E. R. Willeke, « The Inkubook Experience: A Tale of Five Processes », in *2009 Agile Conference*, 2009, p. 156-161.

[17] N. Nikitina et M. Kajko-Mattsson, « Developer-driven big-bang process transition from Scrum to Kanban », in *Proceeding of the 2nd workshop on Software engineering for sensor network applications - SESENA '11*, 2011, p. 159.

[18] F. Kinoshita, « Practices of an Agile team », in *Proceedings - Agile 2008*

Conference, 2008, p. 373-377.
[19] M. Ikonen, « Leadership in Kanban software development projects: A quasi-controlled experiment », in *Lecture Notes in Business Information Processing*, 2010, vol. 65 LNBIP, p. 85-98.
[20] P. Middleton et D. Joyce, « Lean software management: BBC worldwide case study », *IEEE Trans. Eng. Manag.*, vol. 59, nº 1, p. 20-32, 2012.
[21] C. M. Shinkle, « Applying the dreyfus model of skill acquisition to the adoption of kanban systems at software engineering professionals (SEP) », in *Proceedings - 2009 Agile Conference, AGILE 2009*, 2009, p. 186-191.
[22] Software Engineering Standardscommittee, « IEEE Standard for a Software Quality Metrics Methodology », *IEEE Std 1061-1998*, vol. 1998, 2009.
[23] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, et L. G. Votta, « A controlled experiment in maintenance comparing design patterns to simpler solutions », *IEEE Trans. Softw. Eng.*, vol. 27, nº 12, p. 1134-1144, 2001.
[24] M. Ali et M. O. Elish, « A Comparative Literature Survey of Design Patterns Impact on Software Quality », *2013 Int. Conf. Inf. Sci. Appl.*, p. 1-7, 2013.
[25] J. McCAll et etal, « Factors in Software Qaulity », *Three Vol. NTIS AD-A049-OI4, AD-A049-OI5, AD-A049-05*, 1977.
[26] T. Yingjie, C. Chuanliang, et Z. Chunhua, « AODE for source code metrics for improved software maintainability », in *Proceedings of the 4th International Conference on Semantics, Knowledge, and Grid, SKG 2008*, 2008, p. 330-335.
[27] M. O. Elish et K. O. Elish, « Application of TreeNet in predicting object-oriented software maintainability: A comparative study », in *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, 2009, p. 69-78.
[28] Y. Zhou et H. Leung, « Predicting object-oriented software maintainability using multivariate adaptive regression splines », *J. Syst. Softw.*, vol. 80, nº 8, p. 1349-1361, 2007.
[29] W. Li et S. Henry, « Object-oriented metrics that predict maintainability », *J. Syst. Softw.*, vol. 23, nº 2, p. 111-122, 1993.
[30] V. G. Teodorovici, « Practical object-oriented design in ruby », *ACM SIGSOFT Softw. Eng. Notes*, vol. 39, nº 1, p. 48-48, 2014.
[31] I. Gorton, *Essential Software Architecture, Second Edition*. 2010.
[32] M. Keeling, *Design It!: From Programmer to Software Architect*. 2017.
[33] J. F. Dooley, *Software Development, Design and Coding*. 2017.
[34] M. Lotz, « Waterfall vs. Agile: Which is the Right Development Methodology for Your Project? » [En ligne]. Disponible sur: https://www.seguetech.com/waterfall-vs-agile-methodology/.
[35] E. Brechner, *Agile Project Management with Kanban*. 2015.
[36] A. Ejsmont, *Web Scalablity for Startup Engineers*. 2015.
[37] « Wikipedia Git ». [En ligne]. Disponible sur: https://fr.wikipedia.org/wiki/Git.

Liste des tableaux

Liste des figures

ANNEXE

1. **RESTFul** : (**Representational State Transfer**) Architecture des services web ou les interactions avec ces derniers se font avec des jeux d'instructions HTTP pour manipuler les ressources représentées par le service.
2. **JSON** : (**Javascript Object Notation**) format de fichier alternative au **XML,** lisible par les utilisateurs. De plus il est léger et native avec **Javascript**.
3. **UDP** : (**User Datagram Protocol**) : Un protocole réseau alternative au **TCP,** il est utilisé pour les communication qui nécessitent de la vitesse et tolèrent les pertes des paquets.
4. **RPC** : **(Remote Procedure Call)** Architecture des service web similaire au **RESTFul,** où les jeux d'instruction HTTP représentent des commandes de manipulation et non des ressources.
5. **Laravel** : Framework **MVC** développé en **PHP**, offre plusieurs module de gestion des applications web pour faciliter le développement https://laravel.com/
6. **MVC** : **(Model View Controller)** Architecture de software populaire les applications web. Elle permet de séparer les responsabilités, où **Model** gère les données, **View** gère les vues et enfin le **Controller** fais l'interaction entre les deux autres.
7. **ReactJS** : Framework **Javascript** de création des interfaces utilisateur créer par **Facebook Opensource.** Il favorise la composition des éléments ou chaque élément dans une page web est un composant avec une logique, balisage en **JSX** et du style en **CSS**
8. **JSX** : Syntaxe **XML** qui ressemble à du **HTML,** mais qui s'écrit dans un fichiers **Javascript**
9. **CSS** : **(Cascading Style Sheet)** le langage utilisé pour décrire la présentation des pages web.

Résumé

Dans chaque établissement étant public ou privé la gestion de sécurité des accès aux ressources est une préoccupation d'un haut degré, néanmoins sa gestion a toujours été une tâche compliquée. Notre recherche est de proposer une étude depuis la conception jusqu'à l'implémentation d'un système d'information évolutif, sécurisé et extensible de base, qui sera étendu en plusieurs applications. Ou vient notre extension détaillée dans ce mémoire ; Une application de monitoring et contrôle des terminaux de serrures connectés. Notre travail va porter sur les différentes modèles d'architecture et de développement des systèmes d'information extensible et comment entreprendre le cycle de vie des systèmes d'informations. De plus nous allons détailler les bonnes pratiques de la programmation orienté objet en implémentant le principe **SOLID** au cours du développement de ce projet. Comme nous aborderons le développement des interfaces web avec les Frameworks de pointe tel que React. A travers notre cas d'étude, nous avons tenté de répondre à notre questionnement et à démontrer l'intégralité des étapes nécessaires pour aboutir à un système stable, sécurisé et extensible qui pourra être pris comme référence pour des projets futurs.

Mot clés : Sécurité, Accès, Architecture, Système, Extensible, Modèle d'architecture, Serrures Connecté, Pratique Programmation Orienté-Objet, **SOLID**, Frameworks, Laravel, React.

Abstract

In each organization whether in the public or private sector, security management and access control to the company's resources is crucial and one of the highest priority concern. However, it has always been a tedious task. Our research suggests a roadmap for a design study to its implementation of an information system scalable, secure and extensible, which in turn will serve as a basis for other systems that will inherit its capabilities by extending its modules. In this study we will discuss the design process and extend our system for the first time with our case study; an access control interface to manage connected locks. To achieve this goal we start by going through different architectures models and how to iterate over the life cycle of the system. Moreover, we will discuss the SOLID principle in object-oriented programming, also about developing front-end interfaces using frameworks such as React. Throughout our study, we aim to find answers to our problems and demonstrate the required process for building a stable, secure and extensible system that could help reference for later projects.

Keywords : Security, Access, Architecture, System, Extensible, Architecture Models, Connected Locks, Best Practice Object-Oriented, **SOLID**, Frameworks, Laravel, React

ملخص

في كل منظمة سواء في القطاع العام أو الخاص ، تعتبر إدارة الأمن والتحكم في حماية موارد الشركة أمراً حاسماً وأحد أهم الاهتمامات ذات الأولوية. ومع ذلك ، فقد كانت دائما مهمة شاقة. يقترح بحثنا خارطة طريق لدراسة تصميم لتنفيذ نظام معلومات قابل للتوسع وآمن ، والذي بدوره سيكون بمثابة أساس للأنظمة الأخرى التي سترث قدراته من خلال توسيع وحداته. في هذه الدراسة سنناقش عملية التصميم ونوسع نظامنا لأول مرة مع دراسة الحالة الخاصة بنا. واجهة التحكم في حماية لإدارة الأقفال مرتبطة بشبكة الاتصال. لتحقيق هذا الهدف ، نبدأ بالمرور في نماذج التصميم مختلفة و خطوات العمل خلال دورة حياة النظام. سوف يناقش مبدأ SOLID في البرمجة الشيئية، أيضًا حول تطوير واجهات الواجهة الأمامية باستخدام نظام مثل React. نهدف خلال دراستنا إلى العثور على إجابات لمشاكلنا وإظهار العملية المطلوبة لبناء نظام مستقر وآمن وقابل للتوسع يمكن أن يؤخذ كمرجع إلى المشاريع اللاحقة.

الكلمات الدالة : التحكم صلاحية الدخول , تصميم النظام , نظام قابل للتوسع , نماذج تصميم ، أقفال متصلة , أفضل الممارسات البرمجة **React , Laravel SOLID,**