Architecture

Membres de l’équipe

- Adrian NEGRUTA
- Dorian FORNALI
- Dorian GIRARD
- Pierre-Adrien VASSEUR
- Yannick ASCARI

Introduction

Ce document d'architecture vise à fournir une description détaillée de l'architecture du système de santé que nous développons. Il est destiné aux différents acteurs impliqués dans le projet, notamment les développeurs, les administrateurs système, les décideurs ainsi que des futurs membres de l’équipe.

Son objectif est de définir clairement l’architecture technique et fonctionnelle du système, en expliquant les choix technologiques, les contraintes, et les besoins spécifiques auxquels il répond. En effet, il aborde des aspects essentiels tels que la gestion des utilisateurs, la sécurité, la localisation et le stockage des données, ainsi que la conformité aux réglementations (notamment le RGPD). Il inclut également une analyse des risques pour anticiper et minimiser les défaillances possibles.

En parcourant ce document, les lecteurs pourront comprendre la manière dont le système est conçu pour répondre aux besoins des différents utilisateurs (concepteurs, administrateurs, patients, infirmiers, médecins, familles) ainsi que les solutions apportées aux défis techniques et réglementaires.

Le projet est une plateforme de santé destinée à aider les personnes âgées à rester chez elles tout en étant surveillées. Les données de santé sont recueillies par un capteur sur la personne, ces données sont ensuite envoyées à notre système qui va les traiter et les mettre à disposition des différents professionnels de la santé. L’ensemble du contexte et des besoins sera détaillé plus tard pour une compréhension plus approfondie.

Table des matières

Quelques définitions

Utilisateurs du système

- Concepteur
- Administrateurs
- Infirmier
- Docteur
- Patient (personne âgée)
- Proche du patient

Vocabulaire

Données d'identification: Les données relatives à l'identification de l'utilisateur qui proviennent de données renseignées manuellement par l'utilisateur.

- Nom
- Prénom
- Adresse
- Date de naissance
- Adresse mail
- Téléphone

Données de santé: Les données relatives à la santé de l'utilisateur, qui proviennent de capteurs externes à la plateforme ainsi que de données renseignées manuellement par l'utilisateur.

- Fréquence cardiaque
- Pression artérielle
- Niveau de stress
- Oxygénation du sang
- Rythme du sommeil
- Température corporelle
- Détection de la chute
- Sentiment personnel
    - Fatigue ressentie (texte libre)
    - Stress ressenti (texte libre)
    - Sensation de bien-être (texte libre)
    - Anxiété ressentie (texte libre)
    - Notes (texte libre)

Données d'appareil: Les données relatives aux appareils utilisés pour le suivi de la santé des patients.

- Version de l'application mobile
- Version de l'application de la montre connectée

Users Stories

Cette section détaille l’ensemble des besoins utilisateurs traités par le système.

Administrateur

En tant qu'administrateur de la plateforme, je souhaite :

- Avoir accès aux informations des utilisateurs, docteurs, infirmières comme patients, (Pour les patients, les données sensibles sur sa santé seront chiffrées) et les gérer sur la plateforme afin de pouvoir les gérer (CRUD...).
- Attribuer/modifier les rôles de chaque utilisateur (médecin ou infirmière).
- Identifier le smartphone ou la montre connectée d’un patient et pouvoir connaître la version de son logiciel, et en déclencher la notification de mise à jour.

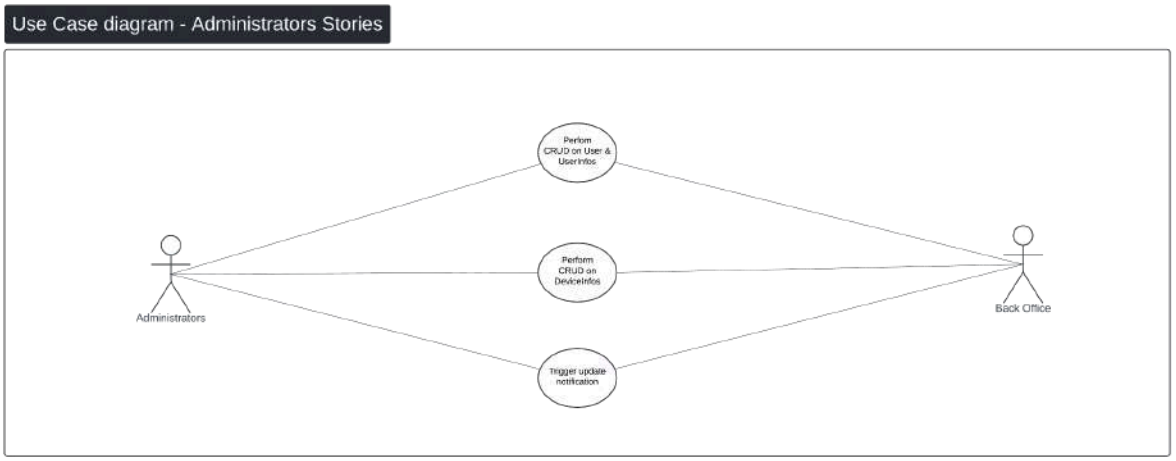

Figure 1. Diagramme de cas d’utilisation - Administrateur

Patient

En tant que patient, je souhaite :

- Recevoir des instructions de l'infirmière lorsque certaines valeurs augmentées/diminuées reçues par les capteurs/appareils (en direct par exemple) sont enregistrées
- Pouvoir, via un formulaire facultatif à remplir une fois par jour, faire des retours sur ma propre santé physique/mentale au corps médical.
- Pouvoir planifier des rendez-vous à domicile avec une infirmière.

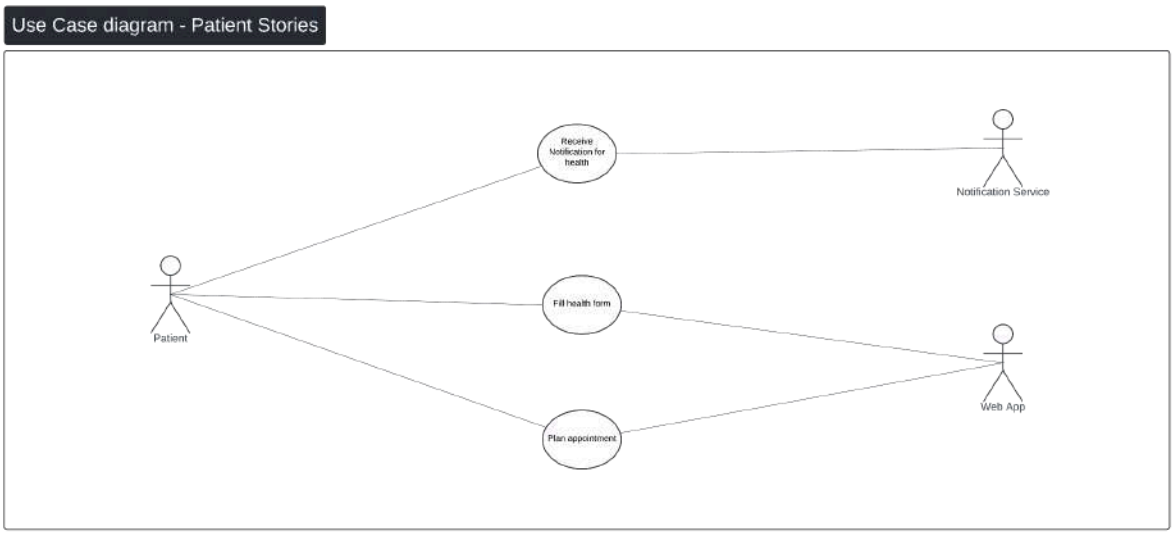

Figure 2. Diagramme de cas d'utilisation - Patient

Infirmier

En tant qu'infirmier, je souhaite :

- Avoir accès à ma liste de patients et leurs proches (et leur coordonnées) afin de pouvoir rechercher et consulter des informations d’identification et de santé sur les patients.
- Être informé via une notification lorsqu'une anomalie a été trouvée dans les données de santé de l'un de mes patients.
- Avoir accès à un bilan de santé global (quantifié) en direct afin d'avoir une idée globale de l'état de santé de la personne âgée.
- Pouvoir voir les rendez-vous planifiés avec les patients
- Planifier des rendez-vous avec mes patients
- Pouvoir faire un rapport au médecin du patient après chaque rendez-vous via un site web
- Pouvoir consulter, modifier ou supprimer les données d’identification de mon compte.

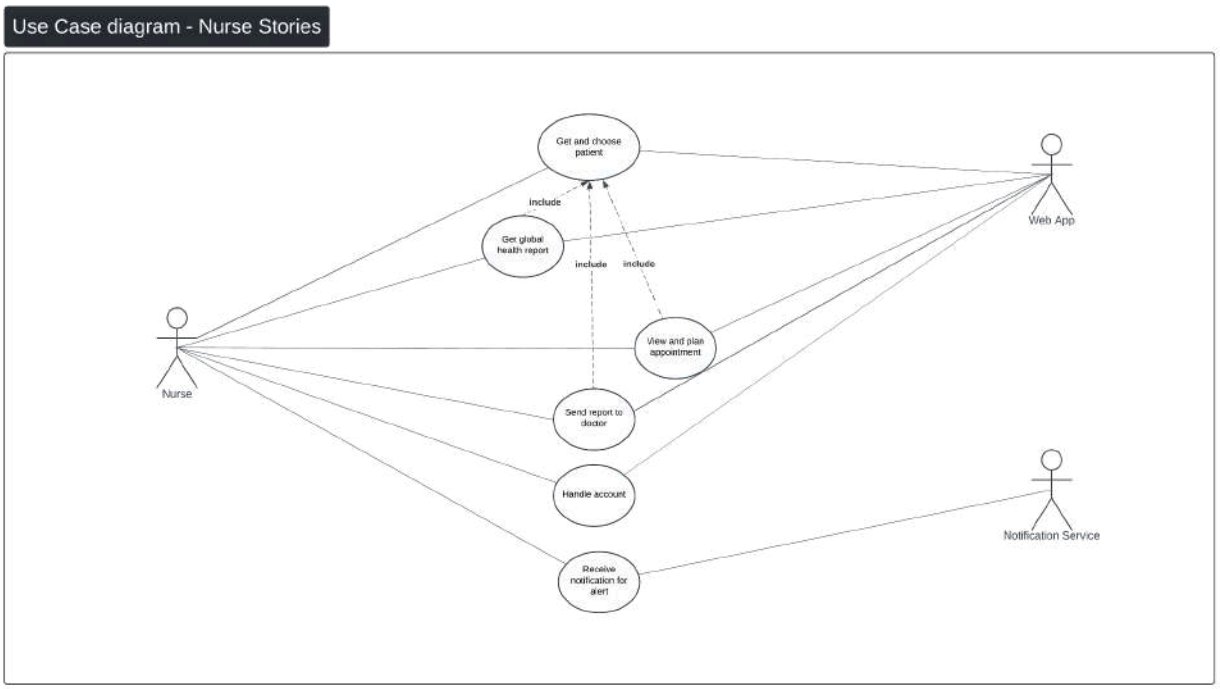

Figure 3. Diagramme de cas d’utilisation - Infirmier

Docteur

En tant que médecin, je souhaite

- Avoir accès à ma liste de patients et leurs proches (et leur coordonnées) afin de pouvoir rechercher et consulter des informations spécifiques sur les patients.
- Avoir un accès complet et illimité à toutes les données de tous mes patients afin de pouvoir les analyser.
- Pouvoir ajouter un patient dans le système afin de pouvoir suivre un nouveau patient.
- Pouvoir associer des infirmières à un patient afin qu'il puisse être suivi par une infirmière
- Pouvoir voir les rendez-vous planifiés avec les patients.
- Planifier des rendez-vous avec mes patients.
- Pouvoir consulter, modifier ou supprimer les données d’identification de mon compte.

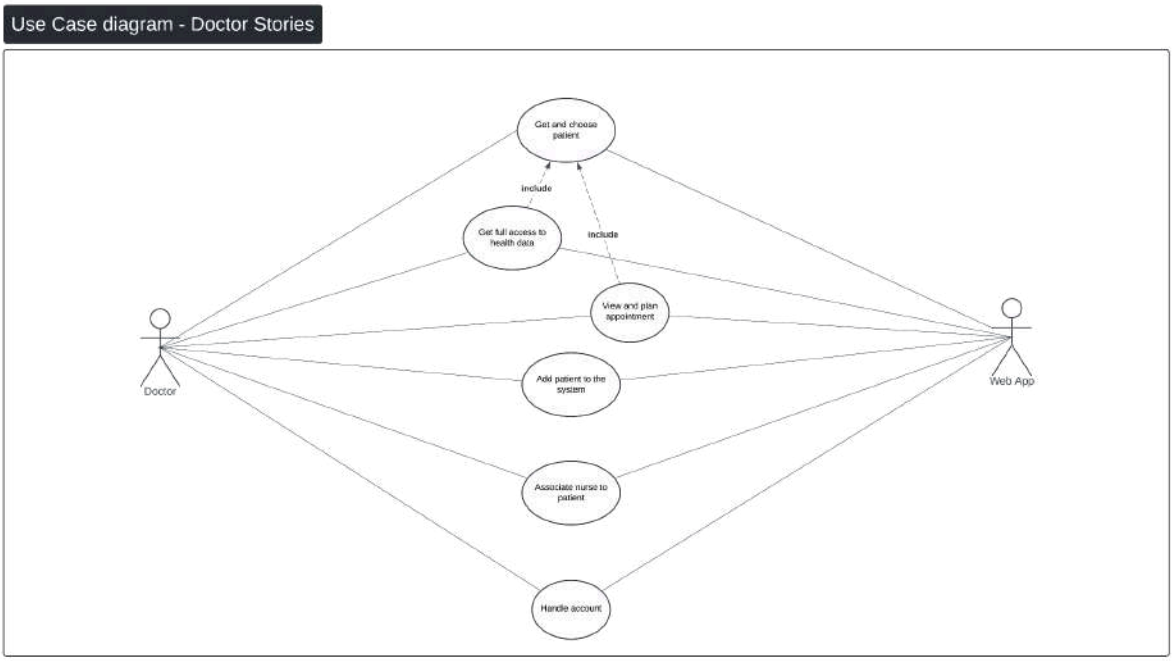

Figure 4. Diagramme de cas d’utilisation - Docteur

Proche du patient

En tant que proche, je souhaite :

- Avoir accès à un bilan de santé global (non chiffré) en direct pour que je puisse avoir une idée globale de la santé de la personne âgée.
- Pouvoir consulter les coordonnées du médecin ou de l'infirmière (email, téléphone) afin de pouvoir les contacter si j'ai besoin d'informations complémentaires.
- Pouvoir consulter, modifier ou supprimer les données d’identification de mon compte.

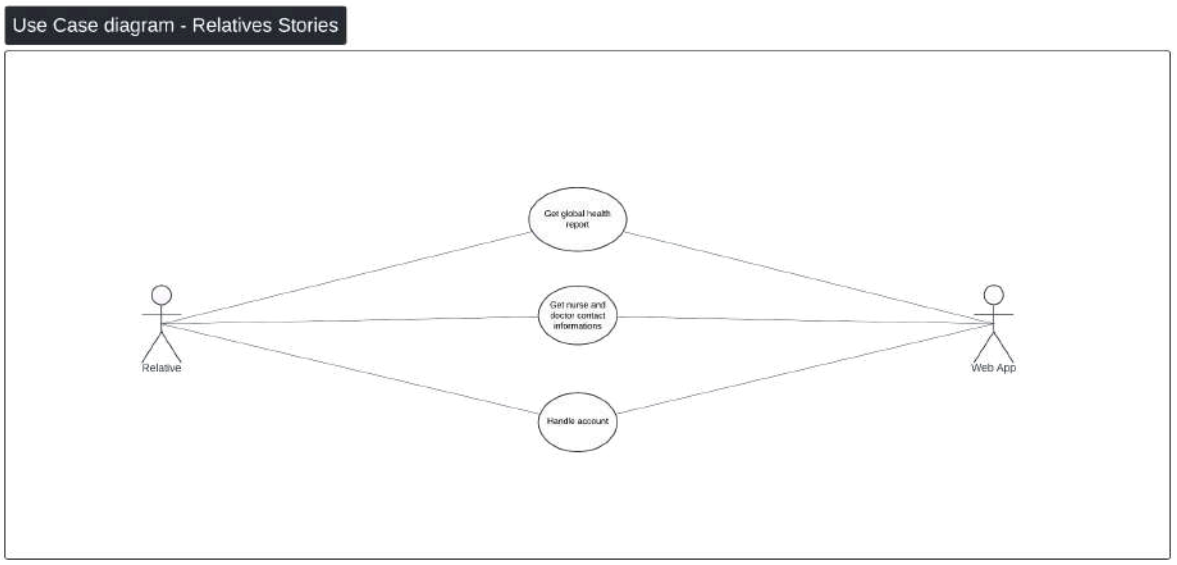

Figure 5. Diagramme de cas d’utilisation - Proche

Concepteur

En tant que créateur du service, je souhaite :

- Avoir un accès au système pour pouvoir effectuer d'éventuelles opérations de débogage, de journalisation, de sauvegarde...
- Pouvoir déployer de manière transparente l'ensemble de la plateforme avec des tests E2E pour effectuer des mises à jour facilement, incluant donc la possibilité de mettre à jour les applications mobiles sur les stores.
- Avoir accès à un tableau de bord des performances et des journaux du système afin de suivre son état de santé et d'identifier les pannes passées comme futures.
- Avoir accès à l'ensemble des sauvegardes du système effectué afin de vérifier leur intégrité, les restaurer…

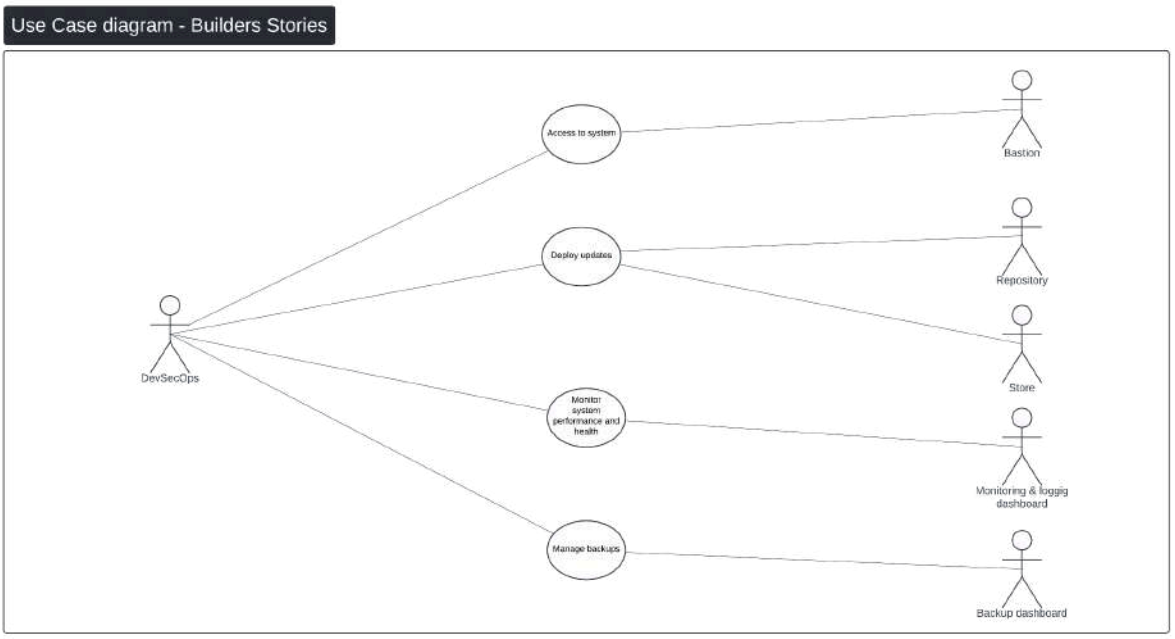

Figure 6. Diagramme de cas d'utilisation - Concepteur

Besoins et contexte de la solution

Le projet consiste à développer une solution e-santé pour aider les personnes âgées à rester chez elles tout en étant surveillées.

Chaque patient est équipé d'une montre connectée qui fait office de capteur. Cette montre est comprise dans le service et est configurée par l'infirmier au départ. La montre est connectée au smartphone du patient et une application est installée sur chaque appareil. L'infirmier s'assure de mettre en place les mises à jour automatiques des applications sur les appareils.

Cette montre connectée collecte des données de santé (fréquence cardiaque, température, détection de chute, etc.) et les envoie à un smartphone. Ces données sont ensuite transmises au système pour être analysées et partagées avec les infirmiers, médecins et proches.

La montre, en coopération avec le smartphone, est capable de détecter une urgence et de la transmettre au système. Les professionnels de santé sont ensuite avertis afin de réagir au plus vite.

Les proches ont accès à un bilan de santé succinct tandis que les professionnels de santé ont un bilan de santé complet avec la possibilité de visualiser l'ensemble de données de santé du patient afin de prodiguer le traitement le plus adapté possible.

Notre système permet de planifier des rendez-vous entre les patients et les infirmiers. Nous laissons aussi la possibilité au patient de remplir un formulaire chaque jour, afin d'exprimer ses pensées et qu'il puisse parler de son propre état de santé, mental comme physique.

L'objectif est de permettre un suivi de santé en temps réel et de gérer les urgences de manière proactive.

Population

Dans la suite de ce document nous partirons du postulat que notre système gère environ 200 000 clients.
En effet, il y a 18,1 millions de personnes âgées en France, nous décidons arbitrairement que notre application suivra 1% de cette population, ce qui représente environ 200 000 personnes.

Transfert et quantité de données

Maintenant, définissons la quantité de données et l'ensemble des données transférées d'un patient au système. Ces chiffres sont une base qui justifient nos choix d'architecture par la suite.

La montre connectée collecte un ensemble de données de santé sur le patient.

Certaines données sont récupérées toutes les minutes :

- Rythme cardiaque

Tandis que d'autres sont récupérées toutes les 10 min :

- Pression sanguine
- Niveau de stress
- Oxygénation du sang
- Température du corps

D'autres données sont récupérées seulement la nuit :

- Rythme de sommeil

Enfin, en cas de chute, on envoie cette donnée au smartphone, qui s'occupera dans un second temps d'alerter le système.

Une donnée sera représentée de la manière suivante :

Donnée	**Type**	**Poids**
Enum	1 int	4 octets
Date / Heure	1 long long	8 octets
Valeur	1 int | string	8 octets

Une donnée pèsera donc 20 octets.

Calculons maintenant le poids des données à transférer par jour pour une personne :

- Rythme cardiaque = 1440 minutes = 1440 * 20 = 28 800 octets = 28,8 Ko
- Données 10 min = 144 minutes = 144 * 20 = 2880 octets = 2.8 Ko * 4 = 11,2 Ko
- Rythme de sommeil = 20 octets = 0,02 Ko
- Urgence = 20 octets = 0,02 Ko
- Donc = 40.1 Ko / 1j / 1p

On obtient alors environ 40.1 Ko par jour par personne de données générées.

Calculons maintenant le poids des données à transférer par jour pour le système entier en se basant sur notre postulat de base de 200 000 utilisateurs :

- 200 000 * 40.1 = 8020000 Ko par jour = 8 Go par jour de données dans le système

On obtient alors environ 8 Go de données à transférer du smartphone à notre système par jour pour l'ensemble des clients.

Architecture technique

Cette section détaille l'architecture technique du projet. Elle comporte l'ensemble des choix techniques et architecturaux que nous avons pris. Nous détaillerons d'abord ce qui est relatif à la maison et ensuite ce qui est relatif au cloud.

Pour chacune des parties et sous-parties, nous explicitons le contexte, les besoins, la technologie choisie, la justification avec avantages et inconvénients ainsi qu'un comparatif avec des alternatives.

Maison

Les applications de montre et mobile seront disponible sur les magasins d'application de chaque plateforme (Android, iOS).

Capteur (Smartwatch)

Besoins

- Suivi continu des données (dans la maison ou dehors)
- Détection d'urgence
- Transmission fiables des données
- Non invasif pour le patient (doit être le plus transparent possible pour le patient, ce sont des personnes âgées donc plus c'est complexe moins ça sera adopté)

Contexte

Montre connectée intelligente de type "Apple Watch" ou "Samsung Watch". Elle est capable de mesurer l'ensemble des données de santé (listé plus bas) ainsi que de détecter des urgences (chute, rythme cardiaque anormal…).

Justifications

Avantages à l'utilisation d'une montre connectée plutôt que d'autres capteurs :

- Quasiment transparent pour le patient
- Ne se limite pas seulement à la maison
- Un seul appareil à gérer par patient (moins d'installation, de maintenance…)
- Mesure permanente des données de santé
- Un grand ensemble de capteurs dans un seul appareil

Inconvénients :

- Autonomie de la montre limitée
- Précision des capteurs peuvent être inférieurs à des capteurs spécialisés
- Limité à certains types de données

Technologie

Apple Watch (WatchOS)

- Swift

    - Langage natif pour les appareils Apple.
    - Optimisé, performances élevées, faible consommation d'énergie et intégration profonde avec l'écosystème d'Apple.
    - Utilisation du framework HealthKit pour accéder aux données de santé.

Samsung Watch (WearOS)

- Kotlin
    - Langage natif phare pour les applications Android et WearOS.
    - Rapide, mise à jour, interopérable avec Java
    - Utilisation de l'API Health Platform pour accéder aux données de santé.

Gateway (Smartphone)

Besoins

- Fiabilité de la connexion
- Stockage temporaire
- Gestion des pannes (capable de stocker plus longtemps et de renvoyer si besoin)
- Compatibilité avec les capteurs
- Haute disponibilité

Contexte

Le smartphone stocke les données localement (base de données locales de type SQLite). Chaque jour à une heure précise, le smartphone transfère les données au cloud. Si le smartphone n'a pas de connexion au moment du transfert, l'envoi est retardé jusqu'à ce que la connexion soit rétablie.

En cas d'urgence, le smartphone notifie directement un micro-service dédié du backend cloud.

Justifications

Avantages à l'utilisation d'un smartphone en tant que gateway plutôt qu'un appareil de type Raspberry Pi :

- Facilité de mise en place (une simple application à télécharger)
- Connexion simplifiée avec la montre connectée
- Ne se limite pas seulement à la maison
- Connectivité avancée (Wifi ou réseau mobile)
    - On ne synchronise les données qu'une seule fois par jour, cela représente quelques Mo au maximum par jour, ce n'est donc pas un problème avec les forfaits mobiles actuels.
- Fonctionne sur batterie
    - On ne synchronise les données qu'une seule fois par jour, il n'y a pas de gros traitements qui sont effectués sur le téléphone, la batterie sera donc préservée au cours de la journée. (on imagine qu'une personne recharge son téléphone tout les jours)

Inconvénients :

- Autonomie et usage (autonomie peut être limitée et l'utilisation d'autres services peut affecter le rôle de gateway)

- Dépendance à l'utilisateur (ne doit pas éteindre le Wifi / Réseaux mobiles / Bluetooth)
- Complexité de développement d'une application mobile

Technologie

iOS - Swift

- Natif, meilleure performance et intégration avec l'OS et l'écosystème (montre).
- Stockage local avec Core Data (surcouche SQLite, framework officiel pour la gestion des données sur iOS)

Android - Kotlin

- Langage phare pour le développement d'applications, mise à jour, optimisé et s'intègre parfaitement avec le système android
- Stockage local avec Room (wrapper SQLite, abstraction moderne et simplifié)

Notes

Et si on utilisait une application hybride (une base de code pour 2 applications iOS & Android) ?

- Avantages d'avoir 2 apps différentes plutôt qu'une app hybride :
    - Accès aux APIs spécifiques pour accéder aux données de santé
    - Gestion fine des tâches de fond
    - Performance et consommation d'énergie optimale
- Inconvénients d'avoir 2 apps différentes plutôt qu'une app hybride :
    - Coût de développement
    - Maintenance des projets
    - Technologies différentes (même si Swift et Kotlin ont tendance à utiliser des concepts similaire)

Dans notre cas, il est plus intéressant d'avoir 2 applications différentes. Nos applications mobiles n'ont pas beaucoup de logique métier et le fait d'utiliser les technologies natives nous permet d'avoir un contrôle fin sur nos applications (accès aux APIs de santé, tâche de fin, optimisation de la batterie…)

Cloud

L’ensemble de nos services est hébergé dans un cloud. On y retrouve un ensemble de services connectés via un message broker avec deux bases de données. Cet ensemble forme le backend du projet. On retrouve aussi la partie frontend, composée d’une application web, un back-office et une API Gateway pour faire le pont entre le frontend et le backend. Enfin, il y a aussi des solutions de sécurité, monitoring, de logging et de backuping dans notre cloud. Tous ces éléments sont détaillés dans cette section. Nous parlerons d’abord de l’infrastructure cloud, puis du backend et enfin du frontend.

Dans les parties qui suivent, nous détaillons nos choix par rapport aux types de cloud (public, privé et hybride) en fonction de nos besoins. Ensuite, nous comparons les différents fournisseurs afin de déterminer celui qui est les plus adapté à notre projet.

Quel type de cloud ? Public, privé, hybride ?

Besoins / Solutions	**Cloud Public**	**Cloud Privé**	**Cloud Hybride**	**Solution choisie**
Scalabilité	Très bonne (infrastructure mutualisée et extensible à volonté)	Limité à l'infrastructure matérielle acquise	Bonne, mais dépend du cloud public pour l'extensibilité	**Cloud Public**
Disponibilité	Très élevée, assurée par le fournisseur (SLA 99.9%)	Très élevée mais dépend de la maintenance interne	Bonne, dépend du cloud public pour la disponibilité	**Cloud Public**
Sécurité	Bonne, mais partagée entre plusieurs clients	Très élevée, contrôle total sur l’infrastructure et les données	Très bonne, avec contrôle sur les données sensibles	**Cloud Hybride**
Résilience	Très élevée (redondance automatique, zones multiples)	Bonne, mais nécessite des solutions internes de reprise après sinistre	Très bonne, combinée entre public et privé	**Cloud Public**
Coûts	Réduit (infrastructure mutualisée et services partagés)	Élevé (investissement matériel et personnel dédié)	Variable selon l’utilisation des services publics et privés	**Cloud Public**
Gestion des identités et accès	Bonne, via les services IAM des cloud providers	Très bonne, contrôle total des accès	Bonne, utilisation des services IAM cloud et internes	**Cloud Public**
Interopérabilité	Très bonne, avec une vaste gamme	Moyenne, solutions internes souvent	Bonne, avec compatibilité entre	**Cloud Public**

Besoins / Solutions	Cloud Public	Cloud Privé	Cloud Hybride	Solution choisie
	de services et API standards	plus personnalisées	systèmes publics et privés	
Monitoring	Très bonne, services natifs (CloudWatch, Azure Monitor, etc.)	À mettre en place en interne	Très bonne, via des outils du cloud public et des solutions internes	**Cloud Public**
Déploiement	Simplifié grâce à des outils et services automatisés	Complexe, gestion en interne du matériel et des configurations	Complexe, avec des interactions entre les deux types d'infrastructure	**Cloud Public**
Respect du RGPD	Très bon (choix de la localisation des données pour respecter la réglementation)	Très bon, contrôle total de la localisation et du traitement des données	Très bon, mais nécessite une attention accrue à la compatibilité entre infrastructures	**Cloud Public**

Conclusion : Cloud Public

Le cloud public répond globalement à nos besoins en termes de scalabilité, coût, disponibilité, monitoring, et respect du RGPD. Bien que le cloud privé soit plus sécurisé et personnalisable, il est plus coûteux et moins flexible. Le cloud hybride pourrait être une alternative pour des besoins très spécifiques de sécurité, mais il ajoute une complexité opérationnelle inutile dans notre cas.

Quel cloud public ? AWS, Microsoft Azure, Google Cloud Platform ?

Besoins	AWS	Google Cloud Platform (GCP)	Microsoft Azure	Solution choisie
Scalabilité	Très bonne (maturité et expérience dans le scaling global)	Très bonne (infrastructure réseau de Google)	Très bonne, forte intégration avec les solutions Microsoft	**AWS**
Disponibilité géographique	Très élevée (zones de disponibilité réparties mondialement)	Moins étendue qu'AWS ou Azure	Très élevée (réseau étendu de Microsoft)	**AWS**
Sécurité	Très bonne (nombreux services de sécurité avancée)	Bonne, mais écosystème plus petit	Très bonne, avec des intégrations Microsoft	**AWS**

Besoins	AWS	Google Cloud Platform (GCP)	Microsoft Azure	Solution choisie
Résilience	Très bonne (régions et zones de disponibilité redondantes)	Bonne, moins répandue géographiquement	Très bonne	**AWS**
Coûts	Relativement élevé, mais adapté à la qualité et la maturité des services	Moins cher pour les services basiques	Coût élevé, souvent comparable à AWS	**GCP**
Interopérabilité	Très bonne, supporte une large gamme de standards et d'API	Bonne, mais offre de services plus restreinte	Bonne, surtout dans les environnements Windows	**AWS**
Gestion des identités et accès	Très bonne, IAM avancé	Bonne, mais encore en développement	Très bonne, surtout avec Active Directory	**AWS**
Monitoring	Très bon (CloudWatch et autres services avancés)	Bon, mais moins avancé que CloudWatch	Très bon, mais peut nécessiter des outils tiers	**AWS**
Déploiement	Très bon, services automatisés pour CI/CD	Bon, simplifié, mais moins d'options avancées	Très bon, mais souvent orienté Microsoft	**AWS**
Respect du RGPD	Très bon (choix des régions pour le stockage de données)	Bon, choix de régions moins large	Très bon	**AWS**
Écosystème et communauté	Très étendu, mature, beaucoup de ressources	Plus petit que celui d'AWS	Étendu, mais plus orienté Microsoft	**AWS**

Conclusion : AWS

Nous avons choisi AWS pour sa maturité, sa disponibilité géographique, et son large écosystème. Bien que GCP offre une excellente infrastructure réseau, et que Azure soit bien intégré avec les solutions Microsoft, AWS répond le mieux à nos besoins en termes de scalabilité, interopérabilité, et flexibilité pour les microservices et le monitoring.

Architecture de notre cloud :

En amont de l'ensemble de nos services, nous avons un WAF (Web Application Firewall). Toutes les requêtes passent par ce pare-feu afin d'appliquer une première couche de sécurité à l'ensemble du système.

Pour les DevSecOps, nous avons une machine qui sert de bastion. Elle est seulement accessible depuis le VPN de l'entreprise par SSH et permet d'accéder à l'ensemble de nos services par ligne de commandes.

Nos services sont orchestrés par Kubernetes (le choix de cet orchestrateur est expliqué plus tard). Nous avons donc un Cluster avec un ensemble de Node. On retrouve au moins deux nœuds. Un nœud pour la partie frontend et un nœud pour la partie backend. Le système est donc capable, si besoin, de monter à l'échelle pour supporter la charge en créant des nouveaux nœuds et pods.

En ce qui concerne le déploiement, nous avons un registre d'images. Lorsque nous décidons de mettre en production un ou plusieurs services, nous envoyons notre code sur la branche "Main" du repository correspondant. Cela déclenche ensuite une pipeline qui va s'assurer que le code compile bien, qu'il est testé puis cela va créer une image docker de ce service pour la déposer ensuite sur un registre d'image. Dès qu'une nouvelle image est disponible et fonctionnelle, l'orchestrateur l'utilise dans le système.

Orchestrateur

Le choix de notre orchestrateur est justifié ci-dessous. Nous avons mis en parallèle plusieurs solutions et nous les avons confronté à nos besoins pour faire ressortir la solution la plus adaptée à notre cas.

Besoins	**Kubernetes**	**Docker Swarm**	**Nomad (HashiCorp)**	**Solution choisie**
Gestion des conteneurs	Très avancée (supporte des configurations complexes)	Simple (intégration native avec Docker)	Très avancée (supporte conteneurs et autres workloads)	**Kubernetes**
Scalabilité automatique	Oui (auto-scaling, support de clusters à grande échelle)	Limitée (scalabilité manuelle principalement)	Oui (auto-scaling intégré et horizontal)	**Kubernetes**
Disponibilité	Très haute (conçu pour une haute disponibilité)	Bonne (mais moins robuste à grande échelle)	Haute (robuste même à grande échelle)	**Kubernetes**
Planification et gestion des ressources	Avancée (planification optimisée, gestion	Simple (moins de contrôles sur la planification)	Avancée (algorithmes intelligents de planification)	**Kubernetes**

Besoins	Kubernetes	Docker Swarm	Nomad (HashiCorp)	Solution choisie
	fine des ressources)			
Résilience	Très bonne (tolérance aux pannes, réplication, auto-réparation)	Moyenne (moins de mécanismes de redondance)	Très bonne (réplication et gestion d'erreurs natives)	**Kubernetes**
CI/CD	Très bon support (intégration DevOps, pipelines CI/CD)	Limitée (moins d'intégrations natives)	Très bon support (intégrations CI/CD avec Consul, Vault)	**Kubernetes**
Sécurité / Isolation	Avancée (RBAC, gestion des secrets, isolation réseau)	Moyenne (moins de contrôle granulaire)	Avancée (RBAC, gestion des secrets via Vault)	**Kubernetes**
Monitoring	Très bon (outils natifs et support de Prometheus, Grafana)	Limité (moins d'outils intégrés)	Très bon (intégration avec Prometheus, Grafana, etc.)	**Kubernetes**
Gestion des réseaux	Avancée (politiques réseau, services mesh, sécurité)	Basique (moins de gestion de la sécurité réseau)	Très bonne (politiques réseau avec Consul)	**Kubernetes**
Portabilité	Très bonne (indépendant du cloud, multi-environnements)	Moyenne (moins flexible, orienté Docker)	Très bonne (compatible multi-cloud et on-premise)	**Kubernetes**
Ressources	Consommation élevée (nécessite une infrastructure conséquente)	Faible empreinte (moins de ressources nécessaires)	Faible empreinte (légèreté et flexibilité)	**Kubernetes**

Conclusion : Kubernetes

Nous avons choisi Kubernetes comme orchestrateur pour répondre aux besoins de notre système. Il offre la scalabilité, la résilience, et la gestion avancée des réseaux nécessaires à notre architecture microservices. Bien qu'il soit plus complexe à configurer et à maintenir, ses fonctionnalités étendues et sa portabilité en font la solution idéale pour une infrastructure à grande échelle. De plus, il possède une très forte communauté avec un écosystème important. Docker Swarm est plus simple, mais limité en fonctionnalités avancées, tandis que Nomad pourrait être une alternative intéressante pour des environnements multi-workload ou multi-cloud.

Backend

Dans cette section nous allons présenter l'architecture de notre backend ainsi que nos choix. Nous étudierons donc les besoins, le contexte, notre justification quant à l'utilisation d'une architecture en microservices plutôt que d'autres types d'architecture et enfin les technologies que nous avons utilisées pour chacun de nos microservices. Les détails relatifs aux bases de données, au message broker et aux services annexes (monitoring, logging, backuping) seront présentés plus tard dans ce document.

Contexte

Le backend est divisé en un ensemble de micro-services :

- Micro-service dédié à la gestion des urgences (chutes, anomalie…)
- Micro-service dédié au prétraitement et stockage des données reçues (Type Pipeline)
- Micro-service dédié au traitement des données de santé
- Micro-service dédié à la gestion des utilisateurs et stockage
- Micro-service dédié à la gestion des rendez-vous entre patients et infirmiers
- Micro-service dédié au suivi et à la gestion des capteurs et appareils (smartwatch, smartphone)
- Micro-service dédié à la gestion des notifications envoyé aux différents types d'utilisateurs

Justifications

Nous avons décidé d'utiliser une architecture en micro-services pour les raisons détaillées dans le tableau ci-dessous. Pour être sûr de notre choix, nous avons comparé plusieurs types d'architectures par rapport aux besoins que nous avons identifiés.

Besoins	Architecture Monolithique	Architecture Microservices	Architecture en Couches	Architecture SOA	Architecture Serverless
Haute disponibilité	Limité	Très bonne (services indépendants)	Bonne (système centralisé)	Bonne (mais plus complexe à scaler)	Très bonne (scalabilité automatique)
Scalabilité	Difficile	Très bonne (scalabilité ciblée)	Difficile à étendre	Moyenne (scalabilité par bloc)	Très bonne (scalabilité auto)
Sécurité	Simple à gérer (un seul bloc)	Complexe (chaque service à sécuriser)	Simple (centralisé)	Complexe (gouvernance centralisée)	Complexe (dépendance au fournisseur)
Résilience	Faible (dépendance à une seule instance)	Très bonne (isolations des services)	Faible (panne impacte tout)	Moyenne (services dépendants)	Très bonne (élimination d'infra dédiée)
Traitement en temps réel	Bonne	Très bonne (services dédiés)	Bonne (couches dédiées)	Moyenne (orchestration nécessaire)	Moyenne (traitement par événements)
Flexibilité sur les technologies	Faible (pile unifiée)	Très bonne (choix technologique indépendant)	Faible (technologie imposée)	Moyenne (mais centralisée)	Limité (dépendant du fournisseur)
Stockage de données différentes	Faible (monolithique)	Très bonne (services spécialisés pour structuré et non structuré)	Faible (centralisé)	Moyenne (centralisé mais flexible)	Moyenne (intégration nécessaire)
Coût	Réduit mais limité en capacité	Plus élevé (infrastructure par service)	Moyen	Moyen	Réduit (facturation à l'utilisation)
Déploiement	Simple mais global	Très bonne (déploiement indépendant)	Complexe (global)	Moyen	Très bonne (automatisé par le fournisseur)
Monitoring	Simple	Complexe (nécessite outils avancés)	Simple	Complexe (centralisé mais lourd)	Complexe (fournisseur-dépendant)
Gestion des transactions	Simple (tout est centralisé)	Complexe (transaction distribuée)	Simple	Complexe	Moyenne (orienté événement)

Conclusion : Architecture Microservices

Nous avons décidé d'adopter une architecture en microservices car elle répond le mieux aux besoins critiques de notre backend, notamment en matière de scalabilité, résilience, et flexibilité technologique. Les hautes disponibilités, la capacité de déploiement indépendant, et la gestion de données structurées et non structurées sont des points forts qui surpassent les architectures monolithiques, en couches, SOA, et serverless.

Technologie

Dans cette section, nous allons détailler l'ensemble des microservices avec le besoin pour chaque, le choix de la technologie avec avantages et inconvénients et la comparaison avec une alternative.

Comme vous allez pouvoir le constater en consultant nos choix et comparatifs des technologies, nous avons essayé de rester constant. Nous avons choisi les technologies les plus efficaces selon les informations que nous avons trouvé pour répondre à nos besoins. Lorsque deux technologies étaient similaires et qu'elles répondaient toutes les deux aux besoins, nous avons fait en sorte de prendre toujours la même technologie afin de rester constant et de faciliter le développement et la maintenance par la suite.

Micro-service dédié à la gestion des urgences

Ce tableau compare quelques technologies intéressantes en fonction de nos besoins identifiés.

Besoins	**Quarkus**	**Go**	**Node.js**	**Solution choisie**
Réactivité	Quarkus est bien adapté aux environnements nécessitant une haute réactivité, surtout grâce à son moteur GraalVM qui permet un démarrage rapide et une faible consommation de mémoire.	Go est particulièrement performant pour les systèmes réactifs et concurrents, souvent utilisés pour des services haute performance.	Moins performant pour des cas critiques en raison d'un modèle d'exécution single-threaded, mais offre une gestion simple des I/O.	**Quarkus**
Traitement en temps réel des événements	Très performant pour le traitement d'événements avec sa faible latence, surtout lorsqu'il est compilé en natif avec GraalVM.	Go, avec ses goroutines, est extrêmement efficace pour traiter des événements en temps réel avec très peu de latence.	Node.js peut gérer des événements en temps réel, mais la performance est limitée par sa gestion non-bloquante et single-threaded.	**Go**
Faible latence	Latence optimisée grâce à la	Go est l'un des meilleurs choix	Latence plus élevée par rapport	**Go**

Besoins	Quarkus	Go	Node.js	Solution choisie
	compilation native de GraalVM et à son approche optimisée de gestion des ressources.	pour des systèmes à faible latence grâce à sa conception performante et légère.	à Go et Quarkus en raison de la nature asynchrone et de l'architecture single-thread.	
Très faible temps pour se lancer et se relancer	Quarkus, en mode natif (GraalVM), propose des temps de démarrage extrêmement faibles, ce qui est idéal pour des microservices qui nécessitent des redémarrages fréquents.	Go a des temps de démarrage rapides, mais ils ne sont pas aussi compétitifs que Quarkus en mode natif.	Node.js a des temps de démarrage raisonnables, mais il n'égale pas les performances de Quarkus ou Go pour ce critère.	**Quarkus**
Écosystème / Frameworks	Quarkus bénéficie de l'écosystème Java mature avec de nombreux frameworks disponibles, tout en offrant une performance optimisée.	Moins de frameworks matures et spécialisés par rapport à Java, mais Go bénéficie de bibliothèques légères et optimisées pour les systèmes distribués.	Écosystème très riche en termes de bibliothèques et plugins, mais moins performant pour les cas critiques.	**Quarkus**

Conclusion : Quarkus

Quarkus est idéal pour la gestion des urgences grâce à ses performances optimisées, sa réactivité et son faible temps de démarrage, surtout en mode natif avec GraalVM. Go est une bonne alternative pour ses hautes performances et sa gestion efficace des événements concurrents, mais il manque de frameworks comparé à Java. Node.js, bien que simple à utiliser, est moins performant que Quarkus et Go dans des environnements critiques.

Micro-service dédié au prétraitement et stockage des données reçues

Justifications de l'utilisation d'une pipeline de traitement :

- Vérifier la qualité des données (anomalie, valeurs manquantes)
- Normalisation des formats de données (homogénéité)
- Détection d'événements (les données sortent du flux de base, elles ne sont pas modifiées et sont marquées)
- Filtrer et agréger les données (moyenne du rythme cardiaque, filtrer les données non pertinentes)
- Ajout de métadonnées (ajout de données liés aux utilisateurs…)

Technologie

Besoins	**Apache Storm**	**Apache Kafka Streams**	**Apache Flink**	**Solution Choisie**
Scalabilité	Très scalable, conçu pour gérer de grands flux de données en temps réel avec des topologies distribuées.	Scalabilité forte, adaptée aux flux de données continus et à l'analyse des données en temps réel.	Scalabilité élevée avec gestion de flux à grande échelle et partitionnement.	**Apache Storm**
Résilience	Assure la tolérance aux pannes avec des mécanismes d'acknowledgement et de redémarrage automatique.	Résilience élevée grâce à l'intégration native dans Kafka et à la gestion de l'état des flux.	Très résilient avec la gestion des échecs intégrée et reprise des tâches en cas de panne.	**Apache Storm**
Filtrage et validation	Permet de filtrer, agréger, et valider les données dans un pipeline complexe, idéal pour les traitements en temps réel.	Kafka Streams offre de bonnes capacités de filtrage et d'agrégation avec des fonctions de traitement légères.	Très efficace pour les opérations de filtrage et validation, surtout pour des pipelines de données complexes.	**Apache Storm**
Latence réduite	Conçu pour le traitement de flux en temps réel avec une latence faible mais peut nécessiter des ajustements pour les systèmes très sensibles à la latence.	Latence plus faible que Storm pour les traitements de flux légers mais moins adapté aux pipelines complexes.	Latence ultra-faible dans des configurations optimisées, surtout pour des traitements d'événements massifs.	**Apache Flink**
Extensible	Facile à étendre avec de nouvelles topologies ou composants de traitement supplémentaires.	Kafka Streams peut être étendu facilement avec de nouveaux stream processors, bien qu'il soit plus	Très extensible, capable de gérer des besoins croissants en ajoutant facilement	**Apache Storm**

Besoins	Apache Storm	Apache Kafka Streams	Apache Flink	Solution Choisie
		orienté vers des analyses.	de nouveaux traitements.	

Conclusion : Apache Storm

Nous avons choisi Apache Storm, principalement pour ses capacités de traitement en temps réel et son évolutivité. Malgré sa complexité, il répond mieux que Kafka Streams ou Flink aux besoins de traitement événementiel et de filtrage complexe. Flink pourrait être envisagé dans des scénarios futurs où la latence est une priorité absolue et où le traitement d'événements complexes nécessite une optimisation plus poussée.

Micro-service dédié au traitement des données de santé

Besoins	Go	Java	Solution Choisie
Traitement en flux continu des données	Très efficace pour les traitements en parallèle grâce à la gestion native de la concurrence et des goroutines.	Java est capable de gérer de grandes quantités de données avec des bibliothèques robustes, mais la gestion du flux en temps réel peut être plus complexe et moins optimisée.	**Go**
Faible consommation de ressources	Go est extrêmement léger, avec une empreinte mémoire réduite et une consommation CPU optimisée.	Java est plus lourd, avec une consommation plus importante de ressources, notamment en mémoire, bien que des optimisations soient possibles.	**Go**

Conclusion : Go

Nous avons choisi Go pour le micro-service de traitement des données de santé en raison de sa performance et légèreté, idéales pour le traitement en flux continu avec faible latence. Java, bien que plus mature pour les processus complexes, est trop gourmand en ressources pour ce service.

Micro-service dédié à la gestion des utilisateurs,

Micro-service dédié à la gestion des rendez-vous entre patients et infirmiers,

Micro-service dédié au suivi et à la gestion des appareils,

Micro-service dédié à la gestion des notifications envoyées aux différents types d'utilisateurs

Pour les micro-services listés ci-dessus, bien qu'ils traitent de domaine métier très différent nous avons identifié des besoins très similaires entre les services et nous avons donc pris le partie de ne mettre qu'un tableau récapitulatif et de choisir la même technologie pour tous afin de rester cohérent.

Besoins	Quarkus	Spring Boot	NestJS	Solution Choisie
Gestion des utilisateurs et des rôles	Prise en charge des fonctionnalités CRUD avec Panache (simplification ORM). Prise en charge de REST facile à configurer.	Large support pour les fonctionnalités CRUD via JPA/Hibernate, bien intégré pour les architectures orientées services.	API CRUD facile à mettre en place avec TypeORM et Prisma pour la gestion des données.	**Quarkus**
Haute disponibilité et scalabilité	Performant et optimisé pour une scalabilité native avec le support d'exécutions serverless, réduisant le temps de démarrage et la consommation de mémoire.	Très robuste pour les environnements de production, mais avec une empreinte mémoire plus élevée que Quarkus et un temps de démarrage plus long.	Bonne scalabilité, particulièrement adapté aux microservices Node.js, mais moins performant que Quarkus et Spring Boot pour les charges intensives.	**Quarkus**
Sécurité et gestion de l'authentification	Bon support des standards de sécurité comme OAuth2 et JWT avec Keycloak intégré. Bonne compatibilité pour des services sécurisés et performants.	Excellente sécurité via Spring Security, très riche en options de configuration mais peut être lourd pour des microservices de petite taille.	Sécurité via Passport.js, adapté pour des configurations plus simples mais nécessite plus de configurations manuelles pour des besoins avancés.	**Quarkus**
Communication interservices et gestion des événements	Supporte Kafka et AMQP avec une intégration native dans le modèle de programmation réactive, parfait pour la communication et la gestion d'événements entre les microservices.	Bonne prise en charge de Kafka, RabbitMQ et autres protocoles via Spring Cloud Stream, mais plus gourmand en ressources.	Utilisation de modules comme EventEmitter ou des solutions comme Redis, mais nécessite plus de configurations pour des systèmes de messagerie avancés comme	**Quarkus**

Besoins	Quarkus	Spring Boot	NestJS	Solution Choisie
			Kafka ou RabbitMQ.	
Facilité de développement et rapidité de déploiement	Outil de développement en continu avec un déploiement rapide. Optimisé pour les environnements cloud et serverless, simplifiant les cycles de développement et déploiement.	Forte communauté et bonne documentation, mais plus complexe à configurer pour des environnements cloud natifs ou serverless.	Développement rapide grâce à une architecture modulaire, mais moins optimisé pour des environnements nécessitant un démarrage rapide et une faible consommation de mémoire.	**Quarkus**

Conclusion : Quarkus

Quarkus se distingue par sa légèreté et son optimisation pour les environnements cloud, tout en offrant des outils performants pour la gestion des utilisateurs, la sécurité, la communication interservices et une haute disponibilité.

Bases de données

Contexte

Notre système sera composé de deux bases de données différentes pour des usages différents. On retrouvera une première base de données SQL pour stocker nos données structurées (informations utilisateurs, informations appareils, gestion des rendez-vous…). Il y aura aussi une base de données NoSQL pour le stockage des données de santé. Plus précisément nous utiliserons une base de données NoSQL de type Time Series car l'ensemble de nos données de santé sont horodatées.

Justification

SQL (Relationnelle) pour les données structurées :

- Avantages :
  - Modèle relationnel (relations complexes entre les données)
  - Transactions ACID (assure la cohérence des données même lors d'opérations multiples ou complexes)
  - Outils et écosystème mature
  - Bonnes performances pour les requêtes relationnelles et les transactions complexes
- Inconvénients :
  - Scalabilité limitée (plus difficile à mettre à l'échelle horizontale par rapport aux systèmes NoSQL)
  - Moins flexible (les changements dans la structure des données nécessitent des modifications complexes du schéma)

NoSQL (Non-Relationnelle) pour les données de santé :

- Avantages :
  - Scalabilité et flexibilité (conçu pour un volume massif de données et une mise à l'échelle horizontale facile)
  - Adapté aux données semi-structurées ou non-structurées (idéal pour les données qui varient dans leur format et leur fréquence)
  - Performances élevées (optimisé pour les lectures/écritures rapides)
- Inconvénients :
  - Moins de support pour les transactions complexes (pas aussi robuste que SQL en termes de cohérence forte)
  - Manque de relations entre les données (ne gère pas les relations complexes aussi bien qu'un système SQL)

Technologie

Base de données SQL pour les données structurées

Besoins	PostgreSQL	MySQL	MariaDB	Solution choisie
Haute disponibilité	Offre des solutions robustes comme le	Bonne réplication master-slave,	Supporte aussi la réplication, mais les outils ne sont pas	PostgreSQL

Besoins	PostgreSQL	MySQL	MariaDB	Solution choisie
	clustering et la réplication	moins avancée que PostgreSQL	aussi riches que PostgreSQL	
Sécurité	Fortes fonctionnalités de sécurité, y compris les transactions ACID, chiffrement des données et gestion des permissions	Basique mais fiable pour les besoins légers, moins riche en fonctionnalités de sécurité	Similaire à MySQL, avec quelques améliorations sur la gestion des permissions	PostgreSQL
Consistance des données	Transactions ACID garantissant la consistance des données	Transactions ACID supportées, mais moins performant pour les requêtes complexes	Transactions ACID et performances légèrement améliorées par rapport à MySQL	**PostgreSQL**
Performances des requêtes	Très performant pour les requêtes complexes, surtout avec des données structurées et semi-structurées (types JSON, etc.)	Excellentes performances pour les lectures simples, moins optimisé pour les requêtes complexes	Meilleure performance que MySQL dans des environnements à forte charge	**PostgreSQL**
Scalabilité	Scalabilité verticale très efficace, mais la scalabilité horizontale nécessite des techniques avancées comme le partitionnement	Scalabilité simple, mais limité pour les très gros volumes de données	Meilleure scalabilité que MySQL, mais encore limitée par rapport à PostgreSQL pour des besoins complexes	**PostgreSQL**
Gestion de gros volumes de données	Peut gérer de gros volumes avec des techniques avancées comme le partitionnement et l'indexation efficace	Performant pour des volumes de données moyens, mais limité pour des données complexes ou volumineuses	Performances légèrement meilleures que MySQL pour des gros volumes, mais en dessous de PostgreSQL	**PostgreSQL**
Données de structure différentes	Supporte les données JSON et semi-structurées, avec des fonctionnalités avancées	Moins performant pour les données semi-structurées, ne supporte pas aussi bien les types de données complexes	Support des types JSON amélioré par rapport à MySQL, mais moins robuste que PostgreSQL	**PostgreSQL**

Conclusion :

PostgreSQL est privilégié pour ses transactions avancées, sa gestion des requêtes complexes et son extensibilité. MySQL et MariaDB sont des alternatives adaptées aux systèmes moins complexes, mais moins performantes pour des environnements exigeants en haute disponibilité et gestion de gros volumes de données.

Base de données NoSQL pour les données de santé

Besoins	InfluxDB	MongoDB	Cassandra (Apache)	TimescaleDB	Solution choisie
Haute disponibilité	Oui, dans les versions distribuées, mais nécessite une configuration avancée	Oui, avec des fonctionnalités de réplication et de sharding	Excellente haute disponibilité grâce à la réplication et au partitionnement distribué	Oui, via PostgreSQL pour une configuration en cluster	**InfluxDB**
Sécurité	Basique, peut nécessiter des configurations supplémentaires pour un usage critique	Bonne sécurité, options pour le chiffrement et le contrôle d'accès	Bonne sécurité, mais souvent complexe à configurer	Sécurité héritée de PostgreSQL, avec des options avancées	**TimescaleDB**
Consistance des données	Eventual consistency (ne garantit pas de transactions ACID complexes)	Consistance flexible, options pour une consistance forte ou éventuelle	Eventual consistency, mais configurable pour des niveaux de consistance élevés	Consistance forte grâce aux garanties ACID de PostgreSQL	**TimescaleDB**
Performances des requêtes	Excellentes pour des requêtes sur les séries temporelles et des analyses de données rapides	Moins optimisé pour les séries temporelles, mais efficace pour des requêtes basées sur des documents JSON	Bonne pour les lectures simples, mais moins performant pour les requêtes temporelles avancées	Bonnes performances pour les séries temporelles, mais peut être plus lent qu'InfluxDB pour des opérations massives	**InfluxDB**
Scalabilité	Scalabilité horizontale, particulièrement pour les opérations d'écriture	Très scalable grâce au sharding et aux réplicas	Excellente scalabilité horizontale et idéal pour des applications distribuées	Scalabilité limitée à celle de PostgreSQL, moins adapté pour des besoins massifs de type Time Series	**InfluxDB**
Gestion de gros volumes de données	Optimisé pour des écritures rapides et des volumes de données en séries temporelles	Capable de gérer de gros volumes, mais plus adapté aux données semi-structurées	Optimisé pour d'énormes volumes de données, idéal pour des systèmes massivement distribués	Peut gérer de gros volumes mais avec des performances moindres pour les Time Series comparé à InfluxDB	**InfluxDB**
Données de structure différente	Efficace pour des séries temporelles avec des données simples en structure	Très flexible pour des données semi-structurées en JSON	Moins performant pour des données relationnelles et non-structurées, mais flexible dans des environnements distribués	Efficace pour les séries temporelles et capable de supporter des requêtes SQL complexes sur des données structurées	**InfluxDB**

Conclusion : InfluxDB

InfluxDB est idéal pour gérer les données de santé en séries temporelles grâce à ses performances élevées en écriture et son stockage optimisé. MongoDB et Cassandra conviennent mieux aux données flexibles et distribuées, tandis que TimescaleDB offre une consistance ACID utile mais est moins performant pour le traitement en temps réel.

Broker

Contexte

Les micro-services communiquent entre eux via un système d'événements et de souscription à ces évènements.

L'approche dirigée par les événements est intéressante dans le cadre de notre projet, on peut facilement imaginer que le micro-service qui gère le traitement des données détecte soudainement une anomalie, il peut alors déclencher un événement spécial qui sera traité par le micro-service qui gère les urgences. Aussi, certaines technologies permettent une priorité entre les évènements, ce qui est très pertinent avec nos systèmes d'alerte.

Justifications

Quelques solutions possibles :

- API REST : Communication synchrone basée sur des requêtes HTTP, simple à mettre en œuvre mais peut entraîner de la latence en cas de surcharge.
- gRPC : Protocole performant utilisant HTTP/2 et la sérialisation via Protocol Buffers, idéal pour les échanges rapides et à faible latence.
- Event-Driven : Utilisation de systèmes de publication/souscription pour envoyer des événements entre micro-services de manière asynchrone.

Besoins	API REST	gRPC	Event-Driven	Solution Choisie
Faible latence	Performance acceptable mais peut souffrir de latence en cas de surcharge	Très performant avec HTTP/2 et Protocol Buffers	Potentiellement plus lent si surcharge des files	**gRPC** pour latence minimale
Scalabilité	Limitée, peut nécessiter des ressources supplémentaires sous forte charge	Scalable, mais exige un équilibrage de charge en fonction des cas d'usage	Excellente, facilite la gestion des pics avec des files de messages	**Event-Driven** pour meilleure scalabilité
Résilience	Moyenne, la surcharge impacte directement la performance	Haute, mais chaque appel reste dépendant d'une réponse immédiate	Très élevée, les événements asynchrones permettent une meilleure tolérance aux pannes	**Event-Driven** pour tolérance aux pannes
Interopérabilité	Élevée, REST est compatible avec la	Compatible entre systèmes	Très bonne, particulièrement	**Event-Driven** pour plus de flexibilité

Besoins	API REST	gRPC	Event-Driven	Solution Choisie
	majorité des systèmes	supportant gRPC, mais nécessite des configurations spécifiques	pour des systèmes hétérogènes nécessitant un découplage élevé	
Asynchrone	Non	Non	Oui, permet de traiter les événements sans attente	**Event-Driven** pour l'asynchrone
Priorité	Complexe à implémenter, pas de gestion native de priorité	Complexe à implémenter, requiert une gestion explicite dans le protocole	Possible, avec des configurations de files priorisées pour certains microservices selon le contexte	**Event-Driven** pour gestion de priorité

Conclusion : Event-Driven

L'approche Event-Driven est optimale pour répondre aux besoins de communication des microservices, grâce à sa scalabilité, sa résilience élevée, et sa capacité à gérer des messages de façon asynchrone avec des priorités. Bien que plus complexe à implémenter et à déboguer, elle offre la flexibilité et l'interopérabilité nécessaires pour un système distribué et hétérogène.

Technologie

Besoins	Apache Kafka	RabbitMQ	Pulsar	NATS	Solution Choisie
Faible latence	Latence faible mais nécessite des configurations avancées pour optimiser	Performances acceptables pour les systèmes de petite taille	Bonne gestion de la latence, même pour des flux de données élevés	Excellente pour les communications rapides	**NATS** pour une latence minimale
Scalabilité	Très scalable, conçu pour gérer de gros volumes de données	Scalabilité limitée pour les très grands systèmes	Haute scalabilité avec support natif de multi-tenants	Faible scalabilité	**Kafka** pour les volumes importants
Résilience	Haute tolérance aux pannes avec réplication des messages	Résilient, mais moins robuste à très grande échelle	Très résilient, stockage persistant natif	Moyenne	**Kafka** pour tolérance aux pannes
Interopérabilité	Large écosystème et support de nombreuses intégrations	Bonne interopérabilité, support de multiples paradigmes	Écosystème en développement avec des intégrations en croissance	Interopérabilité réduite	**Kafka** pour la variété des intégrations

Besoins	Apache Kafka	RabbitMQ	Pulsar	NATS	Solution Choisie
Asynchrone	Oui, permet un traitement asynchrone des événements	Oui, pub/sub et file d'attente	Oui, supporte des scénarios de streaming et de messagerie asynchrone	Oui, asynchrone	**Kafka** pour la flexibilité asynchrone
Priorité	Possible avec des configurations sur les topics et partitions	Support natif des files prioritaires	Possible, mais configuration complexe	Faible support pour les priorités	**RabbitMQ** pour la gestion de priorité

Conclusion : Apache Kafka

Apache Kafka est le choix optimal pour une architecture de microservices avec des exigences élevées en matière de scalabilité, résilience, et interopérabilité, bien qu'il soit plus complexe à configurer. Il est idéal pour gérer des volumes massifs de données et des scénarios de streaming en temps réel.

Monitoring

Contexte

Afin de suivre les performances et la disponibilité de nos services, nous devons mettre en place un système de monitoring avec un tableau de bord clair et détaillé de l'état de chacun de nos micro-services.

Technologie

Besoins	Prometheus + Grafana	ELK Stack	Datadog	Solution Choisie
Surveillance des services critiques	Bonne détection des anomalies via métriques et alertes, mais sans logs	Bonne détection via logs centralisés	Excellente pour les logs et métriques intégrées	**Datadog** pour les anomalies critiques
Monitoring des performances	Excellente pour les métriques de performance, manque la visibilité sur les traces	Système de logs performant mais centré sur l'analyse historique	Très bon pour les métriques, logs, et traces	**Datadog** pour une couverture complète
Suivi de la disponibilité	Via sondes et métriques ; nécessite	Suivi partiel via logs, moins adapté pour les métriques en temps réel	Très bon suivi en temps réel, avec sondes intégrées	**Prometheus + Grafana** pour la simplicité

Besoins	Prometheus + Grafana	ELK Stack	Datadog	Solution Choisie
	configuration manuelle			
Suivi des données d'événements	Efficace pour surveiller des événements via métriques, moins adapté aux logs complexes	Excellente visualisation des logs d'événements	Très bon pour les événements avec logs et traces	**ELK Stack** pour les événements logués
Suivre le trafic réseau	Peut suivre le trafic de base mais limité pour des analyses détaillées	Suivi basique des logs réseaux	Excellent, avec traçage des services	**Datadog** pour des analyses approfondies
Surveillance des bases de données	Possibilité de surveiller les performances des bases de données via métriques	Surveillance basique via logs	Surveillance détaillée, possibilité de configurer des alertes spécifiques	**Datadog** pour la précision
Gestion des alertes	Configuration manuelle mais précise des alertes	Alertes de base sur logs	Gestion d'alertes intégrée	**Prometheus + Grafana** pour la précision
Conformité RGPD	Open-source, les données peuvent être stockées sur des serveurs contrôlés	Open-source mais gestion des logs complexe	Peut poser problème selon les contrôles, solution SaaS	**Prometheus + Grafana** pour le contrôle
Scalabilité	Scalable mais nécessite configuration supplémentaire	Scalable, mais coûteux en stockage	Excellente scalabilité avec gestion SaaS	**Datadog** pour la flexibilité SaaS
Historique	Pas de gestion longue durée	Bon pour l'analyse historique des logs	Gestion historique des données	**ELK Stack** pour les analyses historiques

Conclusion : Datadog

Datadog offre une couverture complète et intégrée pour les besoins critiques, comme le suivi des services, des performances, du trafic, et la surveillance des bases de données.

Logging

Contexte

Afin d'avoir une idée de ce qu'il se déroule dans notre système, nous avons besoin d'un système de logs avec la possibilité de les analyser. Cela nous permettra, entre autres, de détecter des anomalies de comportement et de mettre en place des alertes.

Technologie

Besoins	Grafana Loki	ELK Stack	Graylog	Solution Choisie
Collecte centralisée des logs	Optimisé pour Kubernetes, collecte centralisée avec labels	Très bon pour la collecte centralisée et agrégation	Collecte et agrégation intégrée	**ELK Stack** pour la puissance
Stockage à long terme	Stockage efficace, mais orienté court/moyen terme	Puissant avec politique de rétention complète	Possible, mais scalabilité variable	**ELK Stack** pour la flexibilité
Filtrage et recherche	Recherche limitée par indexation sur labels	Recherche avancée et rapide sur tout le contenu des logs	Recherche efficace avec ElasticSearch	**ELK Stack** pour la puissance
Configuration d'alertes	Configurable mais basique	Configurable avec des alertes avancées	Alertes configurables et avancées	**Graylog** pour la modularité
Conformité RGPD	Open-source, mais stockage limité pour sécurité renforcée	Possibilité de configuration pour stockage sécurisé	Compatible, nécessite configuration spécifique	**ELK Stack** pour la souplesse
Suivi entre services distribués	Suivi basique avec labels, efficace pour Kubernetes	Très bon suivi interservices avec analyse log complète	Suivi interservices avec ElasticSearch intégré	**ELK Stack** pour la traçabilité
Scalabilité	Très scalable avec peu de ressources	Scalabilité possible mais coûteuse	Bonne scalabilité, dépend du stockage ElasticSearch	**Grafana Loki** pour la légèreté
Sécurité	Basique, compatible RGPD mais pas pour données hautement sécurisées	Sécurisé avec configurations avancées	Sécurisé mais demande plus de ressources	**ELK Stack** pour la robustesse
Visualisation et rapports	Visualisation optimisée via Grafana	Visualisation avancée avec Kibana	Visualisation intégrée et configurable	**Grafana Loki** pour la simplicité

Conclusion : ELK Stack

Bien qu'elle soit plus complexe à mettre en place, la solution ELK Stack répond à la majorité de nos besoins critiques (stockage long terme, recherche, suivi interservices et sécurité) et va globalement plus loin que les autres solutions comparées. Nous avons donc décidé de partir dessus pour notre système de logging.

Backup

Contexte

Afin d'avoir une idée de ce qu'il se déroule dans notre système, nous avons besoin d'un système de logs avec la possibilité de les analyser. Cela nous permettra, entre autres, de détecter des anomalies de comportement et de mettre en place des alertes.

Technologie

Besoins	Veeam Backup & Replication	AWS Backup	Bascula	Solution Choisie
Restauration des données en cas de perte	Restauration granulée : fichiers, bases de données, VMs complètes	Restauration rapide, surtout pour les données AWS	Restauration manuelle, dépend de la configuration	**Veeam** pour la flexibilité
Périodicité des sauvegardes	Sauvegarde régulière et configurable	Planification et automatisation natives	Périodicité personnalisable mais complexité élevée	**AWS Backup** pour la simplicité
Vitesse de restauration	Restauration rapide, en particulier pour les environnements locaux	Rapide dans AWS	Moins rapide, selon l'architecture	**AWS Backup** pour AWS
Stockage externalisé	Multi-cloud et stockage sur site possible	Stockage principalement dans AWS (limité en cas de sinistre dans AWS)	Possibilité d'externalisation manuelle, mais complexité accrue	**Veeam** pour la flexibilité cloud
Chiffrement et sécurité	Chiffrement avancé et options de sécurité	Conformité native et chiffrement, surtout pour données dans AWS	Options de sécurité moins avancées	**AWS Backup** pour la sécurité AWS
Automatisation	Automatisation possible avec des configurations avancées	Automatisation native	Automatisation complexe, nécessitant des scripts	**AWS Backup** pour la simplicité
Rétention des données	Rétention configurable pour sauvegarde à long terme	Rétention flexible avec planification avancée	Rétention limitée à des configurations manuelles	**AWS Backup** pour AWS
Flexibilité des types de sauvegarde	Complète, incrémentielle, différentielle	Majoritairement incrémentielle et complète	Complète ou incrémentielle selon la configuration, mais complexité accrue	**Veeam** pour la flexibilité

Besoins	Veeam Backup & Replication	AWS Backup	Bascula	Solution Choisie
Compatibilité avec les bases de données	Prise en charge de multiples bases de données	Compatible avec les services AWS natifs (RDS, DynamoDB, etc.)	Prise en charge selon configuration, mais pas d'intégration native	**Veeam** pour la compatibilité étendue
Multi-cloud	Multi-cloud supporté	Limité à AWS	Pas d'intégration native multi-cloud	**Veeam** pour multi-cloud

Conclusion : Veeam Backup & Replication

Nous avons décidé d'utiliser Veeam pour sa flexibilité, sa compatibilité multi-cloud, son support de différents types de sauvegarde et sa prise en charge étendue. Notre système étant hébergé chez AWS, nous aurions pu utiliser AWS Backup qui offre des fonctionnalités intéressantes pour nos besoins. Cependant, nous ne souhaitons pas nous lier plus que nécessaire avec un cloud provider, selon les informations trouvés, il est très compliqué de sortir les sauvegardes de l'écosystème AWS si besoin.

API Gateway

Besoins de notre API Gateway

- Gestion de l'authentification et de l'autorisation : La capacité à intégrer facilement OAuth2, JWT, et d'autres mécanismes de sécurité.
- Performance et scalabilité : La Gateway doit supporter un grand nombre de requêtes sans introduire une latence significative.
- Routage et agrégation des API : Possibilité d'agréger plusieurs services dans une seule réponse pour optimiser les interactions avec le frontend.
- Facilité de gestion et configuration : Possibilité de configurer facilement les routes, règles de sécurité et gestion du trafic.
- Monitoring : Doit fournir des métriques et logs pour suivre l'état des services.
- Mise en cache : Pour réduire la charge des microservices en mettant en cache certaines réponses.
- Résilience et tolérance aux pannes : Doit offrir des mécanismes de fallback en cas de panne d'un microservice.
- Facilité d'intégration dans un écosystème cloud : Doit bien s'intégrer avec les services cloud comme AWS, Azure, ou Google Cloud, voir des clouds privés.

Contexte

L'API Gateway est un service qui permet de faire le lien avec les micro-services et l'ensemble des interfaces utilisateurs (App web, App BO). Il fournit une interface unifiée sur laquelle les UI peuvent se connecter.
De plus, ce service intégrera un contrôle d'accès par rôle afin de s'assurer qu'un utilisateur est bien autorisé à accéder à un type de contenu.

Justification à l’utilisation d’une API Gateway par rapport à des alternatives

Voici un tableau récapitulatif des besoins pour l'API Gateway avec les solutions possibles et la meilleure option pour chaque besoin :

Besoins	API Gateway	Communication directe avec microservices	Backend for Frontend (BFF)	Service Mesh	Solution Choisie
Gestion de l'authentification et autorisation	Intégration facile avec OAuth2, JWT	Gestion décentralisée et complexe de la sécurité	Dédié à chaque frontend pour personnalisation	Sécurité principalement en interne mais moins d’authentification fine	**API Gateway**
Performance et scalabilité	Support de nombreux requêtes, mais ajoute une légère latence	Pas de latence supplémentaire mais complexité de sécurité	Conçu pour chaque frontend mais nécessite une réplication	Bonne scalabilité interne, mais complexité élevée	**API Gateway**
Routage et agrégation des API	Peut agréger plusieurs services dans une réponse unique	Impossible : chaque service doit être appelé directement	Peut être adapté à chaque frontend mais duplique la logique	Non conçu pour l'agrégation externe	**API Gateway**
Facilité de gestion et configuration	Gestion centralisée, configuration facile des routes, règles de sécurité	Gestion distribuée à chaque service	Dépend des besoins spécifiques de chaque frontend	Complexe à configurer et à gérer	**API Gateway**
Monitoring	Fournit métriques et logs centralisés pour suivre l'état des services	Pas de monitoring centralisé	Monitoring par frontend, mais décentralisé	Monitoring limité aux microservices eux-mêmes	**API Gateway**
Mise en cache	Mise en cache possible au niveau de la Gateway	Non disponible directement	Possible par backend mais nécessite gestion supplémentaire	Pas de mécanisme de cache externe	**API Gateway**
Résilience et tolérance aux pannes	Offre des mécanismes de fallback en cas de panne d’un microservice	Pas de fallback centralisé	Dépend de chaque BFF individuel	Répartition de charge possible mais fallback complexe	**API Gateway**
Facilité d’intégration	Bonne intégration avec	Intégration manuelle,	Chaque BFF s’intègre	Intégration native avec des	**API Gateway**

Besoins	API Gateway	Communication directe avec microservices	Backend for Frontend (BFF)	Service Mesh	Solution Choisie
dans un écosystème cloud	AWS, Azure, Google Cloud	chaque microservice gère son intégration	individuellement mais augmente la complexité	clouds via service mesh, mais complexe à mettre en œuvre	

Conclusion : API gateway

API Gateway est la solution préférée pour répondre aux besoins d'une centralisation des requêtes, de la sécurité, de la résilience, de la mise en cache, et de la gestion. Elle permet aussi d'intégrer des outils de monitoring, d'authentification et de gestion de trafic de manière centralisée.

Technologie

Voici un tableau récapitulatif des besoins pour l'API Gateway avec les solutions possibles et la meilleure option pour chaque besoin :

Besoins	Kong	NGINX (avec NGINX Plus)	Traefik	Solution Choisie
Gestion de l'authentification et autorisation	Plugins OAuth2 et JWT pour authentification et autorisation avancées	Authentification gérée mais complexe pour les options avancées	Moins de fonctionnalités d'authentification avancée	**Kong**
Performance et scalabilité	Hautes performances grâce à NGINX, extensible pour un trafic élevé	Très performant et scalable, bonne gestion de charge	Bonnes performances mais fonctionnalités limitées	**Kong**
Routage et agrégation des API	Routage centralisé, possibilité d'agrégation avec des plugins	Bon routage mais agrégation limitée sans configurations additionnelles	Limité pour l'agrégation des requêtes	**Kong**
Facilité de gestion et configuration	Configuration centralisée mais complexe à l'initialisation, plugins disponibles	Gestion efficace mais plus complexe pour configurations avancées	Simple à configurer, bonne intégration Kubernetes	**Traefik**
Monitoring	Supporte Prometheus et Grafana pour un monitoring avancé	Peut nécessiter des solutions tierces pour un monitoring complet	Monitoring limité sans configurations tierces	**Kong**
Mise en cache	Possibilité de mise en cache intégrée	Mise en cache possible avec	Mise en cache limitée	**Kong**

Besoins	Kong	NGINX (avec NGINX Plus)	Traefik	Solution Choisie
	pour améliorer les performances	configuration supplémentaire		
Résilience et tolérance aux pannes	Gestion centralisée du trafic et résilience via les plugins	Bonne résilience avec load balancing et failover	Moins de tolérance aux pannes sans configuration manuelle	**Kong**
Facilité d'intégration dans un écosystème cloud	Fonctionne bien avec Kubernetes et les environnements cloud, support multi-protocoles	Bonne intégration mais moins flexible avec les environnements modernes	Intégration native avec Docker et Kubernetes	**Kong**

Conclusion : Kong

Kong est la solution la plus adaptée pour la plupart de nos besoins grâce à sa gestion avancée de l'authentification, ses performances élevées, ses capacités de routage et d'agrégation des API, ainsi que son intégration facile avec les environnements cloud. Les plugins disponibles permettent de personnaliser la solution pour répondre aux besoins de sécurité, de mise en cache, et de monitoring.

Frontend

Justifications sur le fait d'avoir deux frontends différents

Avantages :

- Séparation des préoccupations
- UI / UX à destination d'utilisateurs complètement différents.
- Sécurité
    - App web utilisable par n'importe qui
    - Back office web utilisable par les administrateurs via un VPN de l'entreprise.

Inconvénients ?

- 2 projets à développer et maintenir
- 2 projets à déployer avec des ressources pour chacun

App Web

Besoins de notre Web App :

- Accès sécurisé (authentification, autorisation, transfert des données de santé…)
- Interface intuitive et responsive
- Performante
- Haute disponibilité
- Multilingue
- Accessibilité

Contexte

L’application web permet au patient et à la famille d’avoir un statut global sur la santé de la personne âgée.
Pour les infirmiers et docteurs, l’application web permet de voir des graphiques des données de santé, avoir des rapports sur l’état de santé global des patients, pouvoir exporter les données, gérer les différents patients.

Technologie

Besoins	ReactJS	Vue.js	Angular	Solution Choisie
Accès sécurisé	Peut intégrer des bibliothèques pour l’authentification et l’autorisation ; bonne gestion des données sensibles avec JSX et de nombreux packages de sécurité	Bon support des packages de sécurité, notamment pour l'authentification	Intégration complète pour la sécurité via des modules intégrés et la gestion de l’injection de dépendances	**ReactJS**
Interface intuitive et responsive	UI réactive et flexible grâce aux composants ; large choix de bibliothèques tierces pour améliorer l'interface (Material-UI, Ant Design)	Facile à configurer pour des interfaces réactives et intuitives ; bien adapté aux applications de taille moyenne	Moins flexible ; peut nécessiter plus de code pour les composants réactifs et la gestion des animations	**ReactJS**
Performante	Virtual DOM pour des performances optimisées, bonnes pratiques et support pour les applications de moyenne à grande envergure	Bonnes performances pour les applications petites à moyennes, mais moins performant pour les grandes applications	Performant, mais plus lourd que React ou Svelte en raison du framework plus complet	**ReactJS**
Haute disponibilité	Bon support des librairies pour le caching, monitoring, gestion de l'état global, et haute disponibilité	Moins de packages et de solutions prêtes à l’emploi pour les grandes applications et la haute disponibilité	Bon pour les applications d'envergure nécessitant haute disponibilité grâce aux nombreux outils intégrés (Angular Universal pour SSR, etc.)	**ReactJS**
Multilingue	Bon support via des bibliothèques comme i18next ou react-intl, bien adaptées pour le multilingue	Vue-i18n offre un bon support pour le multilingue ; adaptable pour des projets de taille moyenne	Angular a un bon support intégré pour le multilingue, particulièrement adapté pour les applications à grande échelle	**ReactJS**

Besoins	ReactJS	Vue.js	Angular	Solution Choisie
Accessibilité	Support mature pour les fonctionnalités d'accessibilité, large documentation pour les bonnes pratiques (React Aria, etc.)	Bien adapté pour des interfaces simples et réactives, mais documentation et support moins riches que React	Bon support intégré avec de nombreux modules d'accessibilité	**ReactJS**

Conclusion : ReactJS

ReactJS est la solution la plus adaptée, car elle répond bien à nos besoins de sécurité, de réactivité, de performance, de haute disponibilité, et de multilinguisme. De plus, sa flexibilité et sa documentation riche facilitent l'implémentation d'interfaces accessibles, intuitives et performantes.

Utilisation d'un framework type NextJS par dessus React ?

Avantages :

- Server Side Rendering donc chargement initial rapide
- Static Site Generator pour des pages statiques (Homepage vitrine, pages légales par ex)
- API routes et Server components pour s'assurer que de la logique critique reste côté serveur
- Optimisation des performances (lazy load image, page, components, bundle…)

Inconvénients :

- Complexité (SSR, SSR, API Routes, Server components…)
- Charge serveur accrue (SSR, serveur qui tourne en continue)
- Apprentissage du framework

Dans le cadre de notre Web App, l'utilisation de NextJS est totalement justifiée, nous avons besoin de performances et de disponibilité qui nous seront permis par le SSR et l'optimisation qu'effectue NextJS de base. De plus, certains de pages doivent bénéficier d'un SEO avancée pour promouvoir au maximum notre plateforme (les pages vitrines du type page d'accueil, page légales…)

Back Office

Besoins de notre Back Office :

- Contrôle d'accès
- Interface claire et ergonomique
- Performante (recherche, filtre…)
- Journalisation des actions
- Protection des données critiques des utilisateurs

Contexte

Le back office permet aux administrateurs du système de gérer les utilisateurs (CRUD), voir un ensemble d'informations de contact sur ces derniers. Il permet aussi de gérer les smartwatchs et smartphones (voir les versions utilisées par les utilisateurs…).

Technologie

Similaire à la Web app.

ReactJS nous permettra d'intégrer facilement un système back office avec du CRUD et de personnaliser à souhait l'expérience de nos administrateurs.

Utilisation d'un framework type NextJS par dessus React ?

Dans le cadre de notre Back Office, NextJS n'est pas forcément pertinent. En effet, nous n'avons pas besoin d'hautes performances donc les fonctionnalités de SSR, SSG et les optimisations avancées que permettent NextJS ne seront pas forcément utiles. De plus, nous n'avons pas non plus besoin de SEO comme ce site sera totalement fermé au public.

Architecture fonctionnelle

Cette section détaille l'architecture fonctionnelle du projet. Elle permet de représenter le système afin de répondre aux besoins identifiés précédemment (dans les sections users stories, besoins et contextes…)

Elle contiendra principalement des diagrammes de séquences qui illustreront les flux principaux de notre système. Nous nous concentrerons sur les flux les plus importants du système, ceux qui concernent la partie métier spécifique à ce projet.

Collecte et traitement des données de santé

La montre collecte différentes données de santé à des fréquences différentes. Ces données sont transférées au smartphone qui s'occupe de les stockées localement jusqu'au moment où arrive son heure de synchronisation avec le backend. Cela survient une fois par jour. Si à ce moment là, le smartphone n'a pas de connexion, il met en pause l'action et la rejoue dès qu'une connexion mobile ou wifi est disponible.
Ces données sont donc transférées au DataPipelineService qui s'occupe de les prétraiter et les stocker. Ce service est une pipeline donc le flux de traitement est représenté ci-dessous.

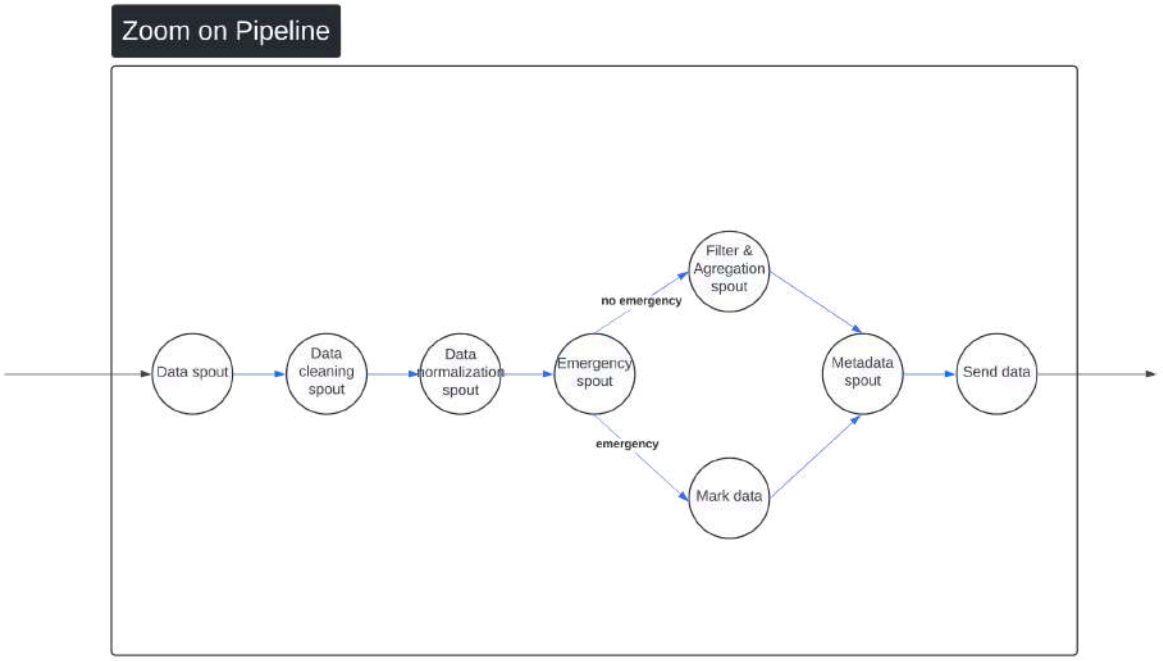

Figure 7. Zoom sur le détail de la pipeline

Bien que la montre et le smartphone vérifie déjà si une urgence survient, nous effectuons une deuxième vérification de plus haut niveau (la montre et le smartphone détecte les urgences immédiates, par exemple une chute, tandis que la pipeline est capable de mettre en parallèle l'ensemble des données reçues de la journée pour détecter un problème, les données sont alors laissées brutes et marquées pour que le personnel de santé puisse vérifier).

Ces données sont ensuite stockées dans la base de données NoSQL et le Health Data Service est alors prévenu que de nouvelles données à traiter sont disponibles. Ce service effectue alors un traitement de haut niveau en mettant en parallèle l'ensemble des données d'un patient sur les derniers jours / dernières semaines. Il génère alors un rapport qui est envoyé aux professionnels de santé. Ce rapport contient les informations suivantes :

- État global de santé du patient
- Un ensemble de valeurs moyennes, minimales et maximales pour les différentes données de santé (par exemple, la valeur moyenne du rythme cardiaque, la valeur minimale sur une certaine période de temps et la valeur maximale sur une période de temps)

En plus de ce rapport, sur la Web App, les professionnels de santé ont aussi accès à l'ensemble des données de santé du patient.

Ce service est aussi capable de notifier un infirmier s'il détecte une urgence qui doit être traitée au plus vite.

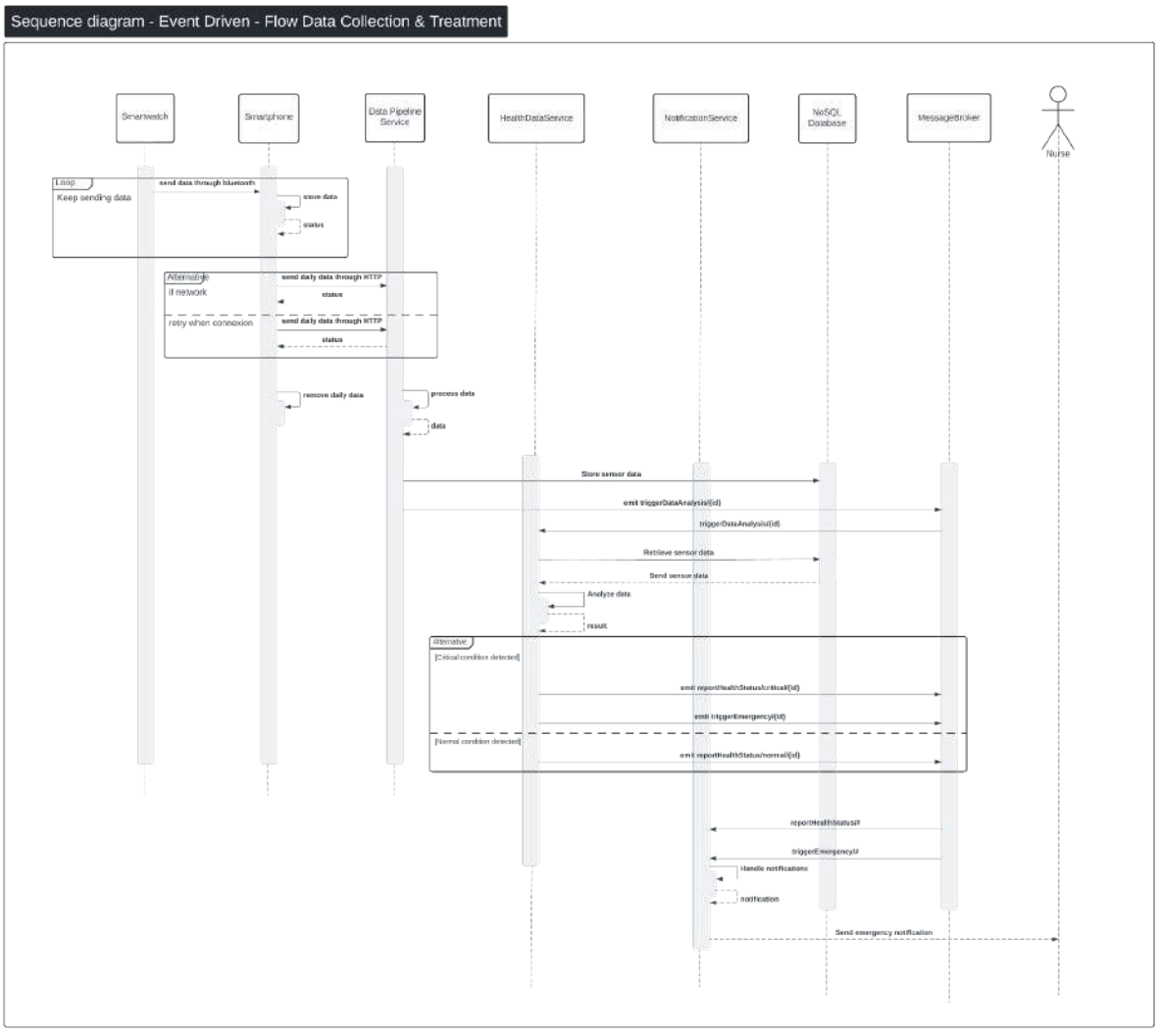

Figure 8. Diagramme de séquence - Flow Collecte et traitement des données de santé

Détection d’une urgence par la montre

Lorsque la montre détecte une urgence (par exemple, une chute du patient sans mouvement durant quelques secondes), elle notifie le téléphone qui va transmettre l’urgence à l’Emergency Service. Ce service va alors vérifier l’urgence reçue et la transférer au Notification Service qui va alors notifier un professionnel de santé.

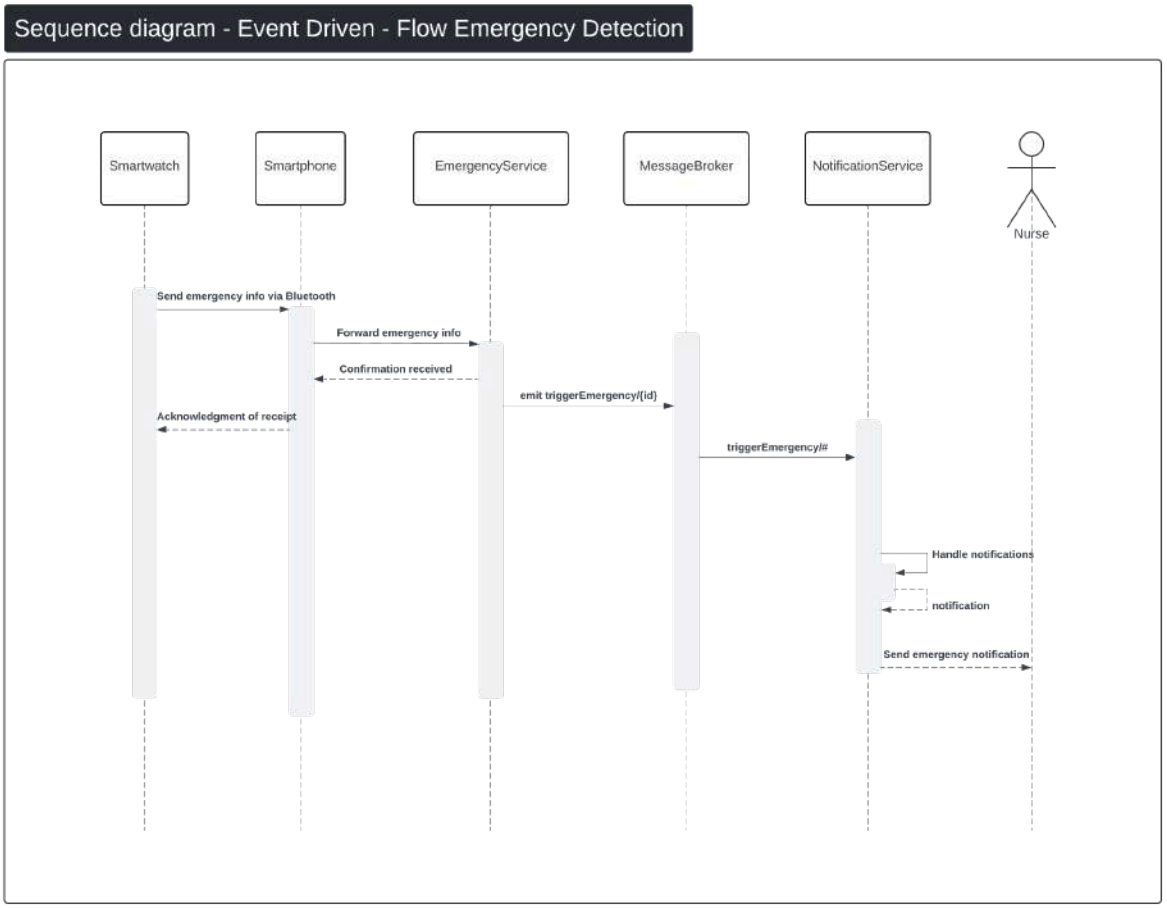

Figure 9. Diagramme de séquence - Flow Détection d’une urgence par les capteurs

Complétion du formulaire par le patient

Une fois par jour, les personnes âgées peuvent compléter un formulaire sur la Web App. Ce formulaire contient un ensemble de champ texte libre (fatigue, stress, anxiété ressenti…). Il permet de suivre l’humeur et son ressenti au fur et à mesure. Une fois ce formulaire complété, il est stocké en base de données et l’infirmier reçoit une notification.
Nous avons décidé de ne pas effectuer de traitement sur ces données. Nous aurions pu demander des valeurs numériques et effectuer des traitements sur ces valeurs afin de déclencher telle ou telle action mais nous avons préféré utiliser des champs textes afin de laisser le patient s’exprimer totalement librement. Il aurait été possible d’appliquer des algorithmes de traitement du langage sur ces textes mais nous préférons garder le côté humain du traitement des ressentis.

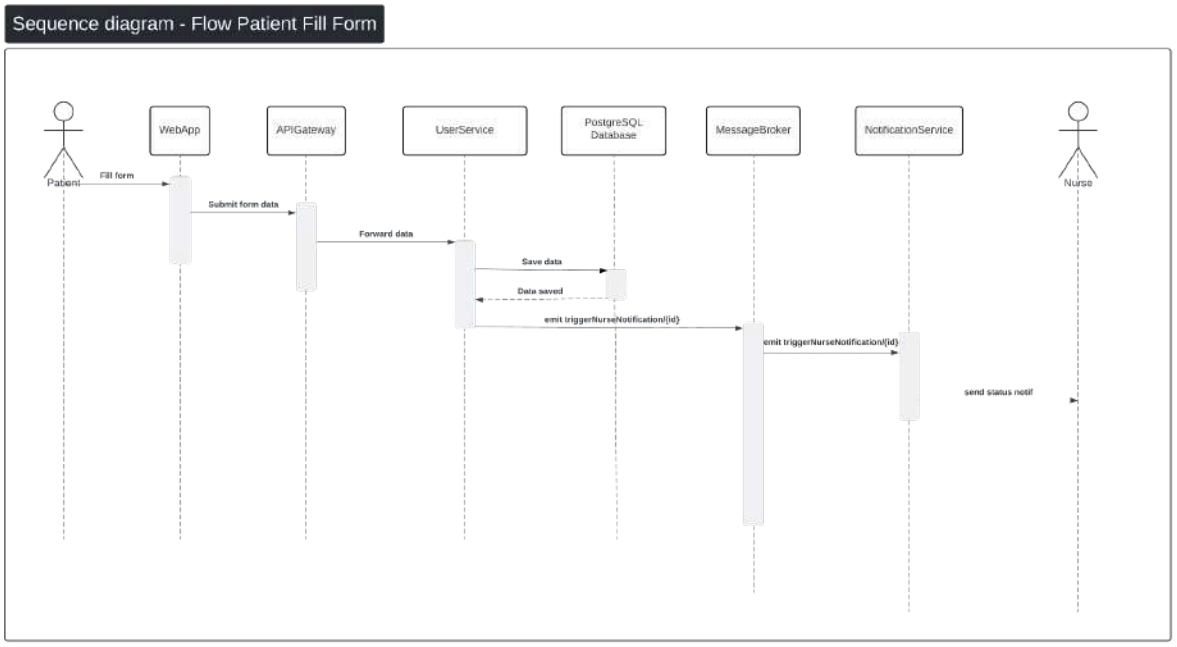

Figure 10. Diagramme de séquence - Flow Patient rempli le formulaire de santé

Réception des données interrompus

Il est possible que notre système ne reçoive plus de données d'un patient. Cela peut survenir pour plusieurs raisons et nous avons mis en place différentes actions en fonction des raisons. Nous allons détailler ci-dessous les différentes raisons possibles et notre réaction.

La montre n'envoie plus de données

Dans la cas où la montre n'envoie plus de données mais que la connexion est maintenue avec le smartphone et qu'elle est portée, nous envoyons une notification à l'infirmier en passant par l'Emergency Service. Pour l'instant, nous laissons le choix à l'infirmier de déterminer s'il prévient directement les urgences. Il serait possible d'imaginer un flow où nous avertissons aussi les urgences dans ce cas là.

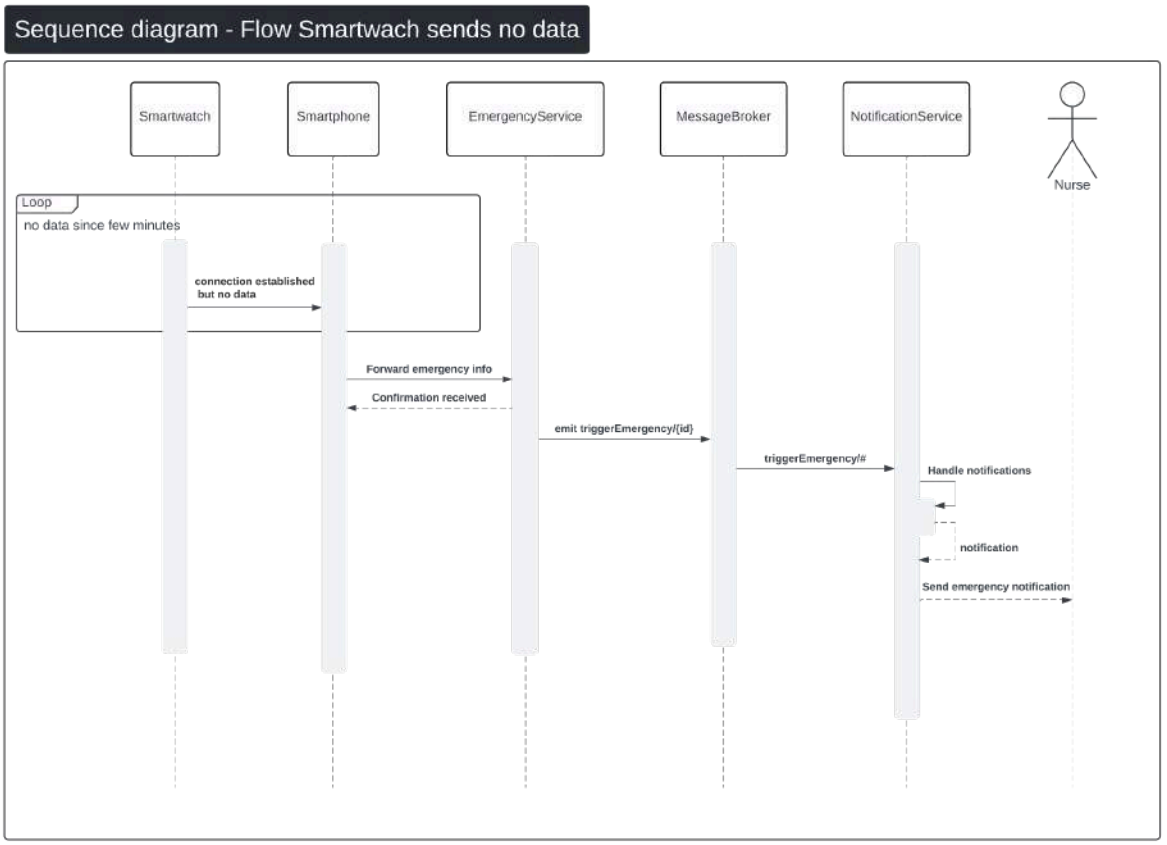

Figure 11. Diagramme de séquence - Flow Montre n'envoie plus données

Le smartphone n'envoie plus de données

Toutes les heures, le micro-service qui s'occupe de la gestion des appareils vérifie la santé des téléphones en faisant un simple ping. Si un téléphone ne répond pas une première fois, le micro-service réessaye plusieurs fois et au bout de quelques essais, il déclenche une notification au proche. Dans ce cas là, on imagine simplement que la personne âgée a oublié de recharger son téléphone et l'on demande à un proche d'aller la visiter pour s'occuper de mettre le téléphone en recharge.

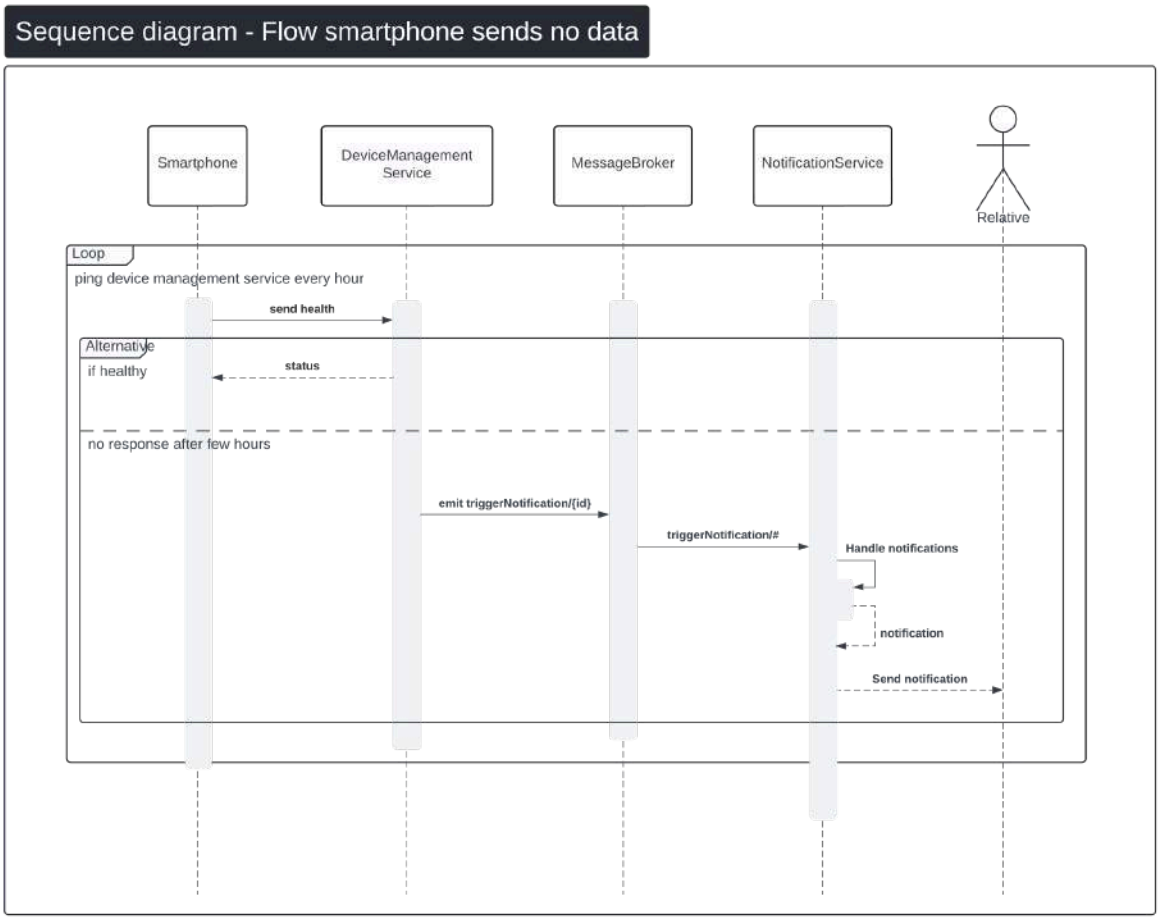

Figure 12. Diagramme de séquence - Flow Smartphone n'envoie plus de données

RGPD

Stockage des données

Types de données collectées

1. Données d'identification
    a. Nom
    b. Prénom
    c. Adresse
    d. Date de naissance
    e. Adresse mail
    f. Téléphone
2. Données de santé
    a. Fréquence cardiaque
    b. Pression artérielle
    c. Niveau de stress
    d. Oxygénation du sang
    e. Rythme du sommeil
    f. Température corporelle
    g. Détection de la chute
    h. Sentiment personnel (remplissage manuel du formulaire)
        i. Fatigue ressentie (1-5)
        ii. Stress ressenti (1-5)
        iii. Sensation de bien-être (1-5)
        iv. Anxiété ressentie (1-5)
        v. Notes (texte libre)
3. Données d'appareil
    a. Version application mobile
    b. Version application montre

Rétention des données

4. Données d'identification
    a. Suppression après 3 ans d'inactivité
5. Données de santé
    a. Suppression après 10 ans d'inactivité
6. Données d'appareil
    a. Suppression après 3 ans d'inactivité

Anonymisation

L'anonymisation n'étant pas obligatoire et nous ne souhaitons pas garder plus que nécessaire les données de nos utilisateurs après avoir atteint leur durée de conversation maximale, nous ne mettons pas en place de processus de pseudonymisation ou d'anonymisation.

Localisation des données

Localisation

Les données sont situées dans des datacenters dans l'Espace Économique Européen (EEE).

Transferts hors UE

Notre cible de marché est située en Europe, nous ne partageons aucune donnée en dehors de l'UE.

Utilisation des données

Finalité du traitement

Les données de santé sont utilisées pour le suivi médical et les alertes d'urgence, tandis que les données utilisateurs sont gérées pour l'accès et la gestion du système.

Consentement des utilisateurs

Les utilisateurs ont signé un document afin de consentir à la collecte et l'utilisation de leurs données pour le bien du service.

Droits des utilisateurs

À tout moment, les utilisateurs peuvent modifier les informations d'identification les concernant. Ils peuvent demander à télécharger les données existantes ainsi que supprimer leur compte.

Sécurité des données

Chiffrement des données

Les données en transit entre les différents services de notre système sont chiffrées avec du TLS. Les données critiques (mot de passe…) sont chiffrées avec un algorithme de chiffrement AES-256.

Contrôle d'accès

Notre système dispose de plusieurs niveaux d'accès :

- Super Administrateur : Concepteur
- Administrateur : Gestion des utilisateurs et des appareils en BO
- Utilisateur : Docteur, infirmier, famille et patient
    - Ont accès à leurs informations et aux informations des utilisateurs liés (un docteur a accès aux informations de son patient)

Lorsqu’un utilisateur se connecte, il doit s’authentifier via un système multi-facteurs.

Journalisation des accès

Le système enregistre les actions critiques sur les données (accès, modification, suppression). Des outils de surveillance sont utilisés pour détecter les activités suspectes.

Localisation des serveurs

Localisation des infrastructures cloud

Nos services sont situés sur des datacenters localisés dans l’Espace Économique Européen (EEE).

Redondance et haute disponibilité

Afin d’offrir la plus haute disponibilité possible, nos données sont répliquées sur plusieurs zones géographiques au sein de l’EEE pour garantir une disponibilité et une continuité en cas de panne, tout en maintenant les données protégées et conformes.

Analyse des risques

Notre analyse des risques se base sur le framework FMEA (Failure Mode and Effects Analysis).
Elle sera découpée en 4 parties :

- Les besoins
- Le champ d'application de l'analyse
- L'équipe de travail pluridisciplinaire
- La matrice d'analyse FMEA

Les besoins

Nous avons besoin d'un système de surveillance de la santé pour les personnes âgées, intégrant des capteurs (montres connectées), la collecte de données en temps réel, la gestion des alertes d'urgence, et la consultation des informations par différents acteurs (infirmiers, médecins, proches). Le système doit être fiable, sécurisé, et respecter les normes de protection des données (RGPD). La disponibilité des services, la rapidité des alertes, et la sauvegarde / restauration des données de santé sont critiques.

Le champ d'application de l'analyse

Nous appliquerons la FMEA au processus de gestion des données de santé, incluant la collecte des données via les capteurs, leur stockage dans les bases de données (SQL et NoSQL), les alertes d'urgence, et les mécanismes de sauvegarde et de restauration des données. Cela couvrira les flux de données, l'intégrité des informations, et les risques liés à la disponibilité des services.

L'équipe de travail pluridisciplinaire

Nous imaginons que l'équipe chargé de réaliser cette analyse des risques est composé des acteurs suivants :

- **Ingénieur Backend** : Responsable des microservices, bases de données et infrastructure cloud.
- **Ingénieur Frontend** : Responsable de l'interface utilisateur pour les patients, proches et professionnels de santé.
- **Expert Sécurité** : Responsable de la conformité RGPD, des protocoles de sécurité des données et des contrôles d'accès.
- **Administrateur Système** : Responsable des sauvegardes, de la restauration des données et de la disponibilité des services.
- **Spécialiste IoT** : Responsable de l'intégration et de la surveillance des capteurs (montres connectées).
- **Responsable produit** : S'assure que le système répond aux besoins des utilisateurs finaux (patients, professionnels de santé).

Matrice d’analyse des risques

Étape du processus	Mode de défaillance potentiel	Effet de défaillance potentiel	Sévérité	Causes potentielles	Occurrence	Contrôles des processus actuels	Détection	Numéro de priorité du risque (Sev * Occ * Det)	Action recommandée
Collecte de données des capteurs	Perte de connexion avec la montre connectée ou le smartphone	Données manquantes pour le suivi de santé	9	Mauvaise connectivité réseau, montre non rechargée…	5	Transfert via Wifi ou Réseaux mobiles selon la disponibilité	5	225	Ajouter un mécanisme de notification pour réagir en cas de perte de données depuis X heures.
Affichage des données sur la WebApp	Chargement lent ou échec du chargement	Mauvaise expérience utilisateur, impossibilité de suivre la santé du patient	7	Surcharge du système	5	Surveillance des performances du serveur	4	140	Mise à l’échelle automatique, mise en cache des données
Stockage des données dans la DB NoSQL	Corruption des données	Inaccessibilité des données vitales	10	Mauvaise gestion des écritures…	3	Sauvegardes récurrentes	3	90	Augmenter la fréquence des sauvegardes, validation des écritures en base
Envoi d’alertes d’urgence	Retard ou absence de l’alerte	Risque pour la santé du patient	10	Défaillance du système de notification	3	Monitoring continue	2	60	Monter un service secondaire si le principale tombe
Sauvegarde des données	Échec de la sauvegarde	Perte de données critiques	9	Problème lors de la sauvegarde ou la restauration	2	Système de sauvegarde programmé	2	36	Vérifications régulières de l’intégrité des données sauvegardées

Sources

Montres :

- https://developer.apple.com/documentation/healthkit
- https://developer.android.com/health-and-fitness/guides/health-services/health-platform?hl=fr

Smartphone :

- https://aws.amazon.com/fr/compare/the-difference-between-web-apps-native-apps-and-hybrid-apps/
- https://developer.apple.com/documentation/coredata
- https://developer.android.com/codelabs/basic-android-kotlin-compose-persisting-data-room?hl=fr#0

Cloud :

- https://www.redhat.com/fr/topics/cloud-computing/public-cloud-vs-private-cloud-and-hybrid-cloud
- https://www.ibm.com/think/topics/docker-swarm-vs-kubernetes
- https://betterstack.com/community/guides/scaling-docker/docker-swarm-kubernetes/
- https://aws.amazon.com/fr/compliance/gdpr-center/
- https://www.redhat.com/en/topics/cloud-computing/public-cloud-vs-private-cloud-and-hybrid-cloud
- https://www.group-dis.com/blog/loffre-cloud/aws-vs-azure-vs-gcp-comparatif-des-offres-cloud
- https://www.digitalocean.com/resources/articles/comparing-aws-azure-gcp

Backend :

- https://www.redhat.com/fr/topics/microservices/what-are-microservices
- https://aws.amazon.com/fr/compare/the-difference-between-monolithic-and-microservices-architecture/
- https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html
- https://www.redhat.com/fr/topics/cloud-native-apps/what-is-service-oriented-architecture
- https://www.redhat.com/fr/topics/cloud-native-apps/what-is-serverless
- https://aws.amazon.com/fr/compare/the-difference-between-soa-microservices/
- https://www.macrometa.com/event-stream-processing/apache-storm
- https://www.devlane.com/blog/should-you-use-golang-advantages-disadvantages-examples
- https://vmsoftwarehouse.com/spring-framework-vs-spring-boot-pros-and-cons
- https://www.redhat.com/en/topics/cloud-native-apps/what-is-quarkus
- https://medium.com/deno-the-complete-reference/quarkus-java-vs-gin-go-hello-world-performance-0a2ec6d92078
- https://medium.com/deno-the-complete-reference/quarkus-vs-go-frameworks-hello-world-performance-03b8eb84dec7

Bases de données :

- https://www.ovhcloud.com/fr/learn/sql-vs-nosql/
- https://db-engines.com/en/system/PostgreSQL
- https://db-engines.com/en/system/InfluxDB
- https://docs.influxdata.com/influxdb/v1/concepts/insights_tradeoffs/
- https://www.ionos.com/digitalguide/hosting/technical-matters/what-is-influxdb/
- https://aws.amazon.com/fr/compare/the-difference-between-mysql-vs-postgresql/
- https://www.ionos.com/digitalguide/server/know-how/postgresql/
- https://cloud.google.com/learn/postgresql-vs-sql?hl=fr
- https://aws.amazon.com/fr/compare/the-difference-between-mariadb-and-postgresql/
- https://medium.com/@servikash/postgresql-vs-mysql-performance-benchmarking-e2929ee377d4

Broker :

- https://learn.microsoft.com/fr-fr/aspnet/core/grpc/comparison?view=aspnetcore-8.0
- https://aws.amazon.com/fr/compare/the-difference-between-grpc-and-rest/
- https://software.land/grpc-vs-kafka/
- https://www.ibm.com/blogs/ibm-france/2023/10/24/point-de-vue-sur-les-architectures-microservices-event-driven/
- https://www.ignek.com/blog/apache-kafka-case-studies-pros-and-cons/
- https://aws.amazon.com/fr/compare/the-difference-between-rabbitmq-and-kafka/
- https://quix.io/blog/kafka-vs-pulsar-comparison
- https://docs.nats.io/nats-concepts/overview/compare-nats
- https://superstreamai1.medium.com/comparing-nats-and-kafka-understanding-the-differences-f08c4479dea6

Monitoring :

- https://medium.com/cloud-native-daily/prometheus-vs-elk-stack-unraveling-the-battle-of-monitoring-and-logging-98e7f17791cd
- https://marclabs.com/prometheus-vs-elk/
- https://medium.com/@squadcast/datadog-vs-prometheus-choosing-the-right-monitoring-tool-for-you-551cddad6d6e
- https://betterstack.com/community/comparisons/datadog-vs-prometheus/
- https://signoz.io/comparisons/newrelic-vs-prometheus/

Logging :

- https://signoz.io/blog/loki-vs-elasticsearch/
- https://opsverse.io/2024/07/26/grafana-loki-vs-elk-stack-for-logging-a-comprehensive-comparison/
- https://medium.com/@disha.20.10/enhancing-kubernetes-logging-capabilities-logging-with-fluentd-fluent-bit-and-loki-part-1-62466f3dfa5e
- https://www.fluentd.org/why
- https://www.cncf.io/blog/2022/02/10/logstash-fluentd-fluent-bit-or-vector-how-to-choose-the-right-open-source-log-collector/
- https://medium.com/@maxy_ermayank/centralized-log-solutions-and-log-shippers-7ff75b699fb
- https://grigorkh.medium.com/what-is-grafana-loki-19a7db694083

Backup :

- https://aws.amazon.com/fr/backup/?nc=sn&loc=0
- https://www.veeam.com/fr/products/veeam-data-platform/backup-recovery.html
- https://www.bacula.org/what-is-bacula/
- https://www.baculasystems.com/blog/aws-s3-backup-software-solutions/
- https://www.veeam.com/products/veeam-data-platform/capability/backup.html

API Gateway :

- https://konghq.com/blog/learning-center/what-is-an-api-gateway
- https://www.designgurus.io/course-play/grokking-system-design-fundamentals/doc/advantages-and-disadvantages-of-using-api-gateway
- https://softwaresennin.medium.com/what-api-gateway-to-choose-kong-gravitee-tyk-or-haproxy-28d7514ad585
- https://www.cloudraft.io/blog/kong-api-gateway
- https://konghq.com/products/kong-gateway
- https://bestcloudplatform.com/choosing-the-right-tool-a-detailed-traefik-vs-nginx-showdown/

Frontend :

- https://medium.com/@VAISHAK_CP/the-pros-and-cons-of-single-page-applications-spas-06d8a662a149
- https://cleancommit.io/blog/spa-vs-mpa-which-is-the-king/
- https://www.browserstack.com/guide/angular-vs-react-vs-vue
- https://medium.com/@reactmasters.in/advantages-and-disadvantages-of-react-js-e6c80b25763b
- https://shakuro.com/blog/svelte-vs-react
- https://medium.com/devsphere/what-is-next-js-and-its-benefits-8b13aab56bfd

RGPD :

- https://www.cnil.fr/fr/passer-laction/rgpd-les-premieres-etapes
- https://www.cnil.fr/sites/cnil/files/atoms/files/referentiel_-_traitements_dans_le_domaine_de_la_sante_hors_recherches.pdf
- https://www.cnil.fr/fr/technologies/lanonymisation-de-donnees-personnelles
- https://www.cnil.fr/fr/les-outils-de-la-conformite/transferer-des-donnees-hors-de-lue

Analyse des risques :

- https://asq.org/quality-resources/fmea