MedHead

PRINCIPES D'ARCHITECTURE

MedHead

PRINCIPES D'ARCHITECTURE

1. Introduction

Ce document décrit les principes de l'architecture avec toute révision des principes et les justifications qui s'y rattachent pour le projet de réalisation de la POC du projet du consortium de MedHead.

2. Contexte

MedHead est un regroupement de grandes institutions médicales œuvrant au sein du système de santé britannique et assujetti à la réglementation et aux directives locales (NHS). Les organisations membres du Consortium utilisent une grande variété de plateformes, de technologies et d'appareils qui souhaite utiliser Java comme langage principal. Une telle plateforme permettra de mettre à profit les similitudes entre ces architectures et technologies de base.

Le projet actuel consiste à réaliser un système permettant de pallier aux risques liés au traitement des lits d'hôpitaux dans des situations d'interventions d'urgence. Pour se faire, il est important de réaliser un POC (Proof of concept) permettant de prouver la faisabilité du projet.

3. Objectifs

Les principaux objectifs de l'entreprise sont les suivants.

1. Tirer parti de la géolocalisation pour permettre aux intervenants de réserver un lit dans l'hôpital compétent le plus proche du lieu d'incident.
2. L'architecture devra être évolutive pour permettre à nos services de se déployer sur diverses régions à travers des villes et des pays donnés.
3. Améliorer les soins dispensés aux patients.
4. Eliminer les risques liés au système d'intervention d'urgence en temps réel.

4. Architecture cible (POC)

Dans le cadre de la réalisation de ce POC, compte tenu des exigences et des contraintes décrites et analysées dans le document «enoncé des traveaux d'architecture», nous avons choisi d'utiliser un architecture en microservices. Ce style d'architecture, répondant aux besoin du projet, présente de nombreux avantages mais aussi des inconvénients.

L'image ci-dessous illustre l'architecture de la POC réalisé, les composants de cette architecture microservices sont décrits dans le document «*enoncé des traveaux d'architecture*».

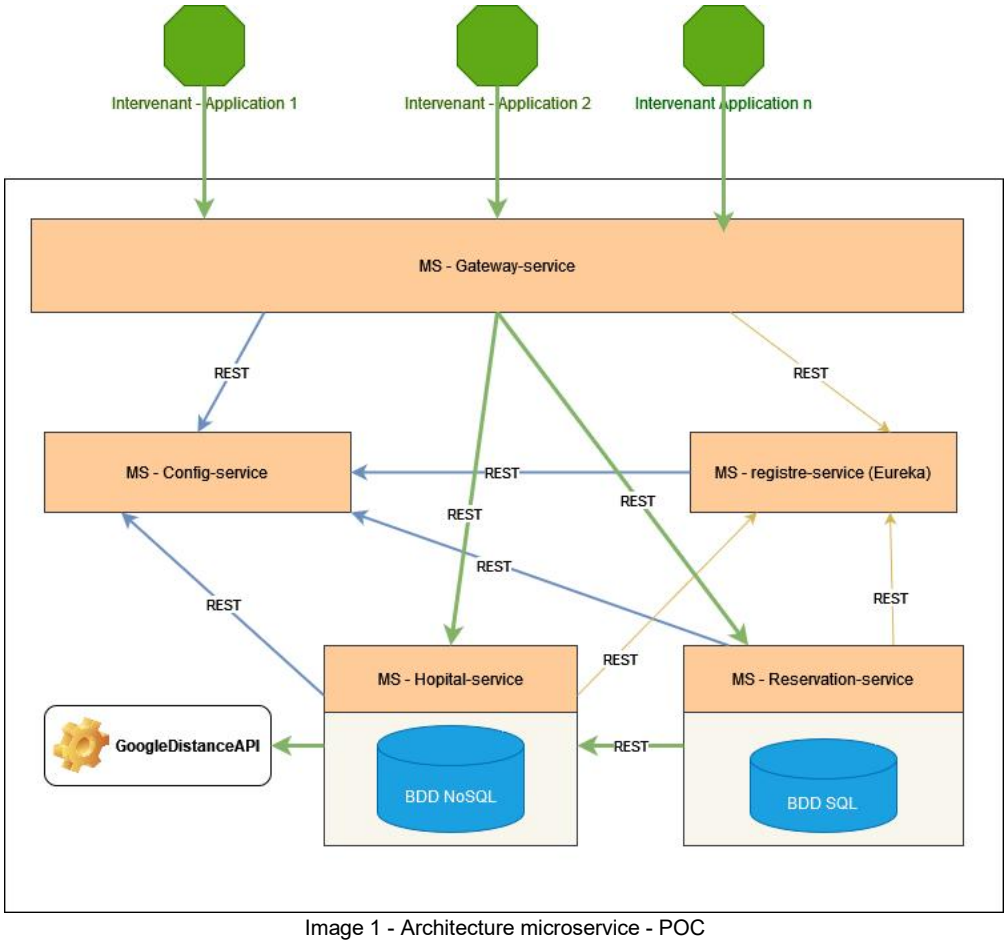

Image 1 - Architecture microservice - POC

4.2. Avantages

1) **Réduction du temps de développement :** Grâce à un développement distribué, plusieurs microservices peuvent être travaillés et développés simultanément.

2) **Evolutivité accrue :** Avec les microservices et leur indépendance en termes de développement et de déploiement, les développeurs peuvent mettre à jour un composant sans que cela n'ait d'incidence sur les autres composants de la solution.

3) **Réduction des pannes** : Les microservices étant indépendants, lorsqu'un élément tombe en panne ou rencontre un problème, l'ensemble de l'application ne cesse pas de fonctionner contrairement aux applications monolithiques. Il est également plus facile d'identifier et de résoudre une panne dans ce type d'ecosystème.

4) **L'adaptabilité de chaque microservice aux outils de travail :** L'indépendance des composants offre la possibilité de paramétrer les microservices avec différents langages de programmation**.** Ainsi, les outils opérationnels utilisés par l'entreprise peuvent être rendus compatibles avec la solution globale.

5) **La flexibilité par rapport au cloud :** Une entreprise qui utilise l'architecture en microservices peut réduire ou augmenter son usage du cloud en fonction de la charge de l'application. L'organisation est ainsi beaucoup plus flexible.

6) **Résistance :** Les microservices, ainsi que la façon dont ils sont abordés, offrent plus de résistance, surtout quand les entreprises sollicitent plus des systèmes. Ce qui équilibre le degré de la charge.

4.3. Inconvénients

1) **Terme « micro » très relatif** : la difficulté est de bien définir la granularité des microservices. Cette granularité est variable d'un microservice à l'autre. Ce n'est pas une bonne idée d'essayer d'obtenir une granularité uniforme pour tous les microservices.

2) **Problème de mise à jour des bases de données cloisonnées** (chaque microservice ayant sa propre base et pouvant être instancié plusieurs fois par le mécanisme de mise à l'échelle). Le mécanisme usuel de transaction de type « commit » n'est plus suffisant.

3) **Défi en matière de débogage, test, déploiement** des applications constituées de microservices. Chaque microservice aura été testé individuellement au préalable, mais ensuite, il faudra bien les tester tous ensemble, de façon automatisée de préférence.

4) **Changements compliqués** à cause des éventuelles dépendances entre les microservices. Les microservices supportent toutefois le versioning et plusieurs versions d'un même microservice peuvent coexister, permettant une migration progressive vers la dernière version.

5) **Application globale moins performante** (latence), car dépendante du réseau (éventuellement moins fiable). Un protocole de communication asynchrone sera souvent préféré à un protocole synchrone, afin d'éviter d'attendre trop longtemps une réponse « immédiate » à chaque requête. De plus, l'usage d'un mécanisme de lecture via une mémoire cache est recommandé, afin d'optimiser la performance.

6) **Sécurité :** Besoin d'authentification, voire de chiffrage, pour diminuer les failles de sécurité du réseau.

4.4. Bonnes pratiques

Le tableau ci-dessous décrit une liste de bonnes pratiques à adopter en matière d'architecture en microservice :

Pratiques	Description
Les microservices doivent communiquer par un protocole indépendant de la plateforme.	Les API Rest permettent de faire communiquer les microservices par un protocole indépendant de la plateforme.
Chaque microservice doit avoir sa propre base de données	Dans notre cas (une base mongodb pour le MS Hopital-service), une base SQL pour le MS (reservation-service).
Un microservice doit avoir une seule responsabilité	MS hôpital-service gère la recherche de l'hôpital MS reservation-service gère la réservation de lits
Utiliser des requêtes asynchrones	
Tracer les requêtes microservice	Centraliser les logs des différents microservices (journalisation).
Scalabilité	Permettre la mise à l'échelle des microservices.
Utiliser un gateway pour accéder aux microservice	Cette passerelle d'api doit être le seul point d'entrée des microservices

5. Principes d'architecture

Un ensemble de principes de l'architecture décrits dans le document «principes d'architecture»(général) fournis avec le projet, ont été pris en compte dans le cadre de la réalisation de ce POC.

Les grands principes d'architecture sont listés dans le tableau ci-dessous :

Principes d'architecture	Description
Principes d'architecture métier	Ces principes incluent : 1. la primauté des principes, 2. maximisation des avantages pour l'entreprise, 3. la conformité aux lois et aux règlement et l'adhésion au serment d'Hipocrate.
Principes de l'architecture informatique (Système, données, solutions, sécurité et opérations)	Ces principes incluent: 1. la continuité des activités des systèmes critiques pour les patients, 2. clarté grâce à une séparation fine des séparations, 3. intégration et livraison continues, 4. tests automatisés précoces, complets et appropriés, 5. sécurité type «shift-left», 6. possibilité d'extention grâce à des fonctionnalité pilotées par des événements.
Méthodologie architecturale et principes de processus	Ces principes incluent : 1. Personnalisation de l'ADM TOGAF 9.2, 2. Référence d'architecture centralisé et organisé comme source de référence, 3. Normes ouvertes convenues pour garantir des normes élevées

	4. Favoriser une culture «learning» avec des preuves de concepts, des prototypes et des spikes.

Dans le cadre de la réalisation de ce POC, le principe architectural que nous avons totalement appliqué est le principes d'architecture informatique (Système données, solutions et opérations).

En effet,

A. **la continuité des activités des systèmes critiques pour les patients** sont assurés et ne nécessite aucun arrêt lors du développement et de l'intégration du «système d'urgence». L'un des avantages de notre architecture est la tolérance aux pannes.

B. **La clarté grâce à une séparation fine des séparations** grâce à l'utilisation de l'architecture en microservices où chaque microservice a une responsabilité unique.

C. **Intégration et livraison continue** grâce à l'utilisation des outils gitHub/Jenkins ainsi que des test automatisés appropriés. Nous avons ainsi crées des pipelines CI/CD qui sont déclenché automatiquement lors d'un commit.

D. **Tests automatisés précoces :** Un ensemble de tests automatisés sont réalisés suivant le principe du BDD (Behaviour Driven Developement) pour garantir la fiabilité du système, notament

   a) Des tests unitaires

   b) Des tests d'intégration

   c) Des tests d'acceptation.

   Lancés automatiquement lors d'un commit, Jacoco configuré dans Jenkins nous permet de visualisés un rapport de test et d'en attester sa couverture par la même occasion. Nous avons aussu utilisé JMeter pour la réalisation des tests de montée en charges.

E. **Sécurité type «shift-left»** : Dès le début, le problème de la sécurité des microservices et du code ont été soulevé et pour cause, nous avons privilégié un développement qui respect les principes SOLID qui sont :

a) Responsabilité unique : classe, fonction ou méthode doit avoir une responsabilité unique

b) Ouvert/fermé : chaque entité (class) est ouverte à l'extension et fermée à la modification.

c) Substitution de Liskov : une instance de type T doit pouvoir être remplacée par une instance de type G, tel que G sous-type de T, sans que cela ne modifie la cohérence du programme

d) Ségrégation des interfaces : préférer plusieurs interfaces spécifiques pour chaque client plutôt qu'une seule interface générale.

e) Inversion des dépendances : Dépendre des abstraction (interfaces) pas des implémentations