



Document de Reporting :

Poc - Recommandation de lits d'hôpitaux

Auteur : Alexandre Boulay

Rôle : Consultant en Architecture Logicielle

Client : MedHead

Table des matières

Introduction	2
But de ce Document	2
Choix Technologiques	3
Serveur Back-End	3
Base de Données	3
Client Front-end	3
Code Coverage	3
Test de Performance	3
CI/CD	3
Normes Respectées	4
RGPD	4
ISO/IEC 25010:2011	4
Architecture MicroServices	4
Respect des Exigences du Projet	5
API REST	5
Interface graphique	5
Pyramide de Tests	5
CI/CD	5
Enseignement de la PoC	5
Choix Effectués	5
Temps de Trajet	5
Problèmes rencontrés	5
Jmeter	5
Éléments Futurs	5
Traduction de l'adresse	5
API privée Google Maps	5
Optimisation de la recherche	6

Introduction

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 actuellement une grande variété de technologies et d'appareils. Ils souhaitent une nouvelle plateforme pour unifier leurs pratiques. La technologie Java est pour eux un socle technique fiable pour ce projet.

But de ce Document

Le but de ce document est de présenter les différents choix et résultats obtenus lors du Proof of Concept de l'application.

Ce Document porte donc sur plusieurs points :

- Prouver que le projet voulu est faisable

- Présenter comment cette démonstration à été faite.
- Justifier les différents choix effectués.
- Présenter les éléments que la mise en pratique du projet a fait apparaître

Choix Technologiques

Serveur Back-End

Dans le cadre de l'API backend mon choix s'est porté sur Spring Boot. Spring Boot est un framework Java très flexible qui permet de développer facilement des API RESTful. Spring permet en quelques clics d'intégrer plein de bibliothèques, par exemple Spring Security qui a permis de sécuriser l'application que se soit l'https mais aussi le système de connexion a token.

Base de Données

La base de données choisie est une base de données H2 car elle est sécurisée et s'intègre très bien avec l'existant. C'est une base de données légère qui permet facilement de stocker les éléments nécessaires.

Client Front-end

Dans le cadre du frontend j'ai choisi Nuxt.js qui est une extension de Node.js avec les avantages de VueJs et de ViteJs. C'est à dire que, comme Spring des modules peuvent très facilement être intégrés, de plus grâce a ViteJs les différents éléments sont chargés à la volée et évitent de recharger les différents éléments.

La bibliothèque d'éléments étendue permet d'aller au-delà du JavaScript classique par exemple dans le code j'utilise fetch pour des requêtes simples et useFetch pour des requêtes plus complexes et asynchrones.

Pour l'affichage nous avons choisi Tailwind qui ergonomise l'interface avec quelques classes html. De plus il s'intègre facilement avec vue.js pour uniformiser le style des pages

Code Coverage

Dans le cadre de l'estimation de code coverage du client backend cela à été fait avec Jacoco qui est le principal outil Java pour fournir ces métriques. Il fournit une page web détaillée vers l'ensemble des différents éléments testés et calcule en plus du coverage les différentes branches qui aurait pu être oubliées.

Test de Performance

Dans le cadre des Tests de performance nous avons utilisé Jmeter car il permet de simuler de grosses charges de travail de manière réaliste et est facilement configurable. Le fait qu'il soit configurable permet de faire des scénarios complexes et d'avoir de bonnes mesures des différents éléments pour optimiser l'API.

CI/CD

Pour l'intégration continue l'outil retenu à été Jenkins car :

- Étant la plateforme CI/CD la plus utilisée, il possède donc une grande communauté d'utilisateurs et un support actif.
- Il intègre nativement l'ensemble des technologies que j'ai choisi (Spring, nodeJS).
- Son système de Multi-Pipeline pour GitHub est vraiment intéressant dans le cas ou plusieurs branches doivent cohabiter comme dans notre projet.

Normes Respectées

RGPD

Dans le cadre de la RGPD l'ensemble des données stockées sur l'utilisateur sont juste son mot de passe encrypté et son identifiant. Une page de contact détaillant les points est mise en place sur le site et nous pouvons effacer l'ensemble des éléments voulu pour l'utilisateur. Dans ce PoC j'ai donc bien pris en compte les éléments suivants de la RGPD:

- Consentement et transparence : Il est présenté à l'utilisateur ce que nous faisons des données et que nous ne stockons que le nom qu'il a rentré et son mot de passe de manière anonymisé
- Minimisation des Données : Comme cité plus haut le minimum d'éléments sont stockés et transmis
- Sécurité des données : l'ensemble des sites sont en https et l'authentification de l'utilisateur est obligatoire pour accéder au contenu
- Droits des utilisateurs : Les utilisateurs sont informés de leurs droits d'accès et de modification/suppression de leurs données.
- Partage des données : Le de la conversation avec les APIs externes l'élément transmis est l'adresse mais ces APIs n'incluent pas de base de données pour la sauvegarde des requêtes.

ISO/IEC 25010:2011

L'application réponds à l'ensemble des contraintes imposées par la norme 25010 :

- Capacité fonctionnelle
- Facilité d'utilisation l'interface est simple et claire avec peu d'éléments
- Fiabilité l'ensemble des éléments résistent aux différents tests aléatoires imposés
- Performance les éléments résistent aux différents tests de charges
- Maintenabilité l'ensemble des éléments déployés sont simples avec des tests et des commentaires prévu pour une évolution facile
- Portabilité - le package généré peut s'exécuter sur l'ensemble des environnements, dans le cadre des démarche à suivre nous envisageons de containeriser cela car le serveur de test actuel ne supportais pas la containerisation

Architecture MicroServices

Le système actuel est un système microService, un seul service est fourni au travers d'une API qui se connecte à d'autres API. Il y a bien un ensemble base de données/serveur par élément désiré.

Respect des Exigences du Projet

API REST

Ce projet respecte la contrainte imposée d'être une api restful car il existe dans un ensemble SOA et respecte l'interface REST :

- Utilisation des méthodes HTTP: POST pour l'enregistrement et le login, GET pour les différentes API de service.
- Ressources identifiées par les urls, ici auth/login, /hospitals ...
- Codes d'états HTTP : 200 réussite, 400 mauvaise valeurs, 401 mauvaise authentification, etc..
- Les échanges de données se font au format JSON*
- L'API est stateless, chaque requête possède les données pour être traitée indépendamment.









Interface graphique

Ce projet comporte une interface graphique simple et ergonomique permettant à un utilisateur de se connecter ou s'enregistrer simplement, puis de choisir sa spécialité voulue puis entrer son adresse et accéder à la liste des hôpitaux proches.

Pour cela Nuxt.js à permis de mettre en place une application web avec du rendu côté serveur, Tailwind a permis de faire la majorité du style des différentes pages et tout cela s'interface sans bug avec l'API.

Pyramide de Tests

MedHead

Element	Missed Instructions	Cov.
 com.ocal.medhead.service		78 %
 com.ocal.medhead.model		83 %
 config		74 %
 dto		92 %
Total	243 of 1 297	81 %

Tests Unitaires

Un ensemble de tests unitaires ont été réalisés sur l'ensemble des éléments d'entités, de configuration et de DTO (Objets de transfert de Données).

Tests d'intégration

Des tests d'intégrations ont été fournis sur l'ensemble des Services et de Register

E2E

Pour les Test E2E nous avons utilisé Postman, l'ensemble des requêtes possibles ont été testé. Utilisations normales, utilisation sans autorisation, ensemble de requêtes aléatoires valides, ensemble de requêtes aléatoires non-valides, requêtes avec éléments manquant.

Test de Stress

Les tests de Stress ont été effectués avec Jmeter, l'ensemble de l'application marche avec des temps valable pour l'ensemble des éléments(requêtes simples, distances à vol d'oiseau) sauf les API externes qui sont détaillées dans la section Enseignement de la Poc-> Problèmes Rencontrés.

CI/CD

L'ensemble du développement continu est déployé sur Jenkins via une pipeline multibranche qui scanne les différentes branches de Code. Une fois la branche dev validée et Pulled sur le main un projet Jenkins est notifié via un webhook et construit l'archive.

Enseignement de la PoC

Choix Effectués

Temps de Trajet

Dans le cadre de cette PoC le choix à été retenu de calculer les trajets en temps de trajet au lieu de distance brute, car pour une ambulance le but est d'arriver le plus vite à sa destination. Dans ce cas, il est nécessaire de questionner des API externes qui ralentissent un peu la vitesse de réponse de l'API mais permettent de réduire grandement le temps de traitement d'une urgence et l'urgence sera belle et bien acheminée vers l'hôpital le plus proche. L'Angleterre possédant différentes îles proches du continent, un patient ayant besoin d'un hôpital sera plus loin d'un hôpital situé sur une île en termes de temps malgré que la distance puisse être moindre, de plus les Hôpitaux du centre de Londres sont parfois difficiles d'accès.

Génération de données aléatoires

Dans le cadre de la PoC les données aléatoires du nombres de lit disponible on été volontairement mis bas pour tester profondément les capacités du système. La majorité des hôpitaux ayant 200 lits de libres en moyenne cela représente un mauvais échantillon de test, ici ils ont été placés entre 0 et 5.

Résultats des Tests

Le Résultat des tests est que l'ensemble est faisable et répond en majorité aux demandes des parties prenantes, l'interface est fonctionnelle et ergonomique. L'ensemble est sécurisé, le client web et l'api sont en https, l'ensemble des authentifications sont sécurisées, les mots de passe des clients sont encryptés pour ne pas pouvoir être lus.

Problèmes rencontrés

Jmeter

Dans le cadre de l'utilisation de Jmeter avec des APIs de mapping externe, le fait de tenter 800 requêtes sur la même interface externe avec les mêmes paramètres fait que l'API de nomination pense que j'effectue un DDOS sur son interface. Le passage sur un ensemble payant ou le déploiement en interne permettra de pallier à ce problème. L'API reste performante sur l'ensemble des requêtes passées avant le Time-out.

Éléments Futurs

Traduction de l'adresse

Afin d'améliorer le service fournit l'interface qui traduit l'ensemble des adresses en coordonnées devrait être présente en local via une API hébergée chez nous. Cela réduirait le temps des requêtes.

API privée Google Maps

Dans le cadre de ce PoC l'API utilisée est une API open source OSRM, le plus efficace serait d'utiliser le leader du marché dans la mesure où il établit des temps de trajet très précis et peut exporter ses routes directement vers le système GPS de l'ambulance. Je pense qu'il est important de garder à l'esprit que quelques minutes gagnées sur un trajet d'ambulance sont plus importantes que quelques millisecondes sur une requête.

Optimisation de la recherche

Certains algorithmes permettent d'optimiser la recherche via coordonnées dans la mesure où le jeu de données est fixe (hôpitaux, routes ..). Dans le cadre où un budget conséquent sera alloué, il serait intéressant d'optimiser les requêtes grâce à ce type d'algorithmes qui intégrerait directement les jeux de données et les traductions d'adresses. Cela permettrait d'avoir besoin de communiquer avec GoogleMaps seulement pour avoir les routes congestionnées via une requête et calculer les trajets sans devoir questionner des services externes pour chaque requête.