RAPPORT DE LA PREUVE DE CONCEPT POUR LE SYSTÈME D'INTERVENTION D'URGENCE DE MEDHEAD

1. Introduction

Le consortium MedHead, composé de grandes institutions médicales britanniques, a identifié la nécessité de moderniser et d'unifier les pratiques liées à la gestion des lits d'hôpitaux en situation d'urgence. Ce rapport présente les résultats d'une preuve de concept (PoC) visant à valider l'architecture cible pour un nouveau système d'intervention d'urgence basé sur PHP Laravel et intégrant diverses technologies modernes.

2. Objectifs de la PoC

Les objectifs principaux de cette preuve de concept (PoC) étaient multiples et visaient à garantir que le système proposé pourrait répondre efficacement aux besoins des institutions membres du consortium MedHead. Les objectifs détaillés sont les suivants :

Validation de l'architecture proposée

- Vérifier que l'architecture choisie répond aux exigences fonctionnelles et non fonctionnelles du système, incluant la gestion efficace des lits d'hôpitaux et la capacité à répondre rapidement en situation d'urgence.
- Assurer que le système est conçu de manière modulaire sous formes de micro-services, permettant ainsi des extensions et des mises à jour futures sans perturbations majeures.
- Évaluer les mécanismes de sécurité intégrés dans l'architecture pour protéger les données sensibles des patients et des hôpitaux.

Test des performances et de la robustesse du système

- Mesurer les temps de réponse des différentes API pour s'assurer qu'ils restent inférieurs à 200 ms dans des conditions normales d'utilisation.
- Évaluer la capacité du système à maintenir des performances acceptables sous des charges élevées, simulant des situations d'urgence avec des pics de demandes (800+ utilisateurs).

Évaluation de l'intégration des technologies choisies avec les systèmes existants

- Tester l'interopérabilité avec les systèmes existants des institutions médicales, même si l'intégration complète n'est pas requise pour cette PoC. Chaque partie doit être pensée et réalisée en micro-services et doit pouvoir s’intégrer à l’existant.

- Identifier les éventuelles difficultés d'intégration et proposer des solutions pour une transition fluide vers le nouveau système.

- Évaluer la qualité de la documentation et les besoins en formation pour les utilisateurs finaux et les administrateurs système.

3. Méthodologie

La méthodologie employée pour cette preuve de concept (PoC) a été structurée en plusieurs étapes clés afin de garantir une évaluation rigoureuse et exhaustive de l'architecture proposée.

Mise en place de l'environnement

Environnement de Développement

Backend - Java Spring Boot

- **Version de Java** : 17 Corretto
- **Framework** : Spring Boot
- **Gestionnaire de dépendances** : Maven
- **Base de données** : H2
- **Test unitaire** : JUnit

Frontend - Nuxt

- **Version de Nuxt** : 3
- **Framework** : Vue.js
- **Gestionnaire de paquets** : yarn
- **Système de styles** : Tailwind CSS
- **Librairie graphique** : Nuxt UI
- **Langage de programmation** : JavaScript

Outils de développement

- **IDE** : IntelliJ IDEA
- **Gestionnaire de versions** : Git

Justification des technologies utilisées

Back-end

Développé en **Java** **Spring Boot**. Ce choix technologique s'explique par plusieurs avantages :

Spring Boot est réputé pour sa robustesse et sa scalabilité, le rendant idéal pour les applications de grande envergure nécessitant des performances élevées. Son écosystème riche, avec de nombreux modules et bibliothèques, facilite le développement rapide et sécurisé d'applications web complexes.

Pour cette PoC, Spring Boot permet également de créer des micro-services légers, facilement déployables et maintenables, ce qui est crucial pour tester rapidement différentes architectures et intégrer de nouvelles fonctionnalités sans perturber l'ensemble du système.

Front-end

Le framework **TS/JS** basé sur **Vue.js**: **Nuxt 3** a été choisi. C'est une technologie moderne qui s'intègre parfaitement avec les APIs du back-end.
Nuxt 3 offre une excellente expérience de développement grâce à sa modularité. Le frameworks Vue facilite la création de composants réutilisables et modulaires, rendant l'interface utilisateur adaptable et extensible. La librairie de composant **Nuxt UI** permet la création de composants graphiques facilement configurables.
Cette technologie permet de créer des interfaces utilisateur intuitives et réactives, essentielles pour ce projet de santé.

D'autres options peuvent également être envisagées pour le front-end.

- **React**, possède un écosystème riche et populaire dans la communauté des développeurs, React est une alternative viable. Il permet la création de composants réutilisables et offre une grande flexibilité.
- **Angular** est un autre choix puissant. Il offre un cadre complet pour le développement d'applications web avec des fonctionnalités intégrées pour les formulaires, le routage et les services, ce qui peut accélérer le développement et garantir une structure de projet cohérente.

Base de données

La base de données Java **H2** a été utilisée pour cette PoC. H2 est une base de données en mémoire qui permet des manipulations rapides et faciles des données de test sans nécessiter une configuration complexe. Son utilisation simplifie le processus de développement et de test en fournissant un environnement léger et performant, ce qui accélère les cycles de développement et permet de se concentrer sur l'implémentation et la validation des fonctionnalités clés sans être freiné par des configurations lourdes.
Pour une PoC, H2 est idéale car elle permet des itérations rapides et des tests fréquents, assurant ainsi une validation rapide et efficace de l'architecture et des composants du système.

Cependant, pour une mise en production, il est préférable d'utiliser une base de données plus robuste et évolutive.

- **PostgreSQL** est recommandée pour son extensibilité, sa conformité ACID et ses fonctionnalités avancées comme les transactions complexes et la gestion des grandes volumétries de données. PostgreSQL est une base de données relationnelle open-source puissante qui convient aux applications de production nécessitant une haute fiabilité et performance.
- **MySQL** est une autre option populaire qui offre de bonnes performances et une gestion efficace des données. MySQL est souvent utilisé dans l'industrie et bénéficie d'un support étendu et de nombreux outils pour la gestion et l'administration des bases de données.

Ces bases de données sont plus adaptées aux environnements de production où la stabilité, la performance et la capacité à gérer des charges de travail importantes sont cruciales.

Jeu de données

Le fichier d'instructions SQL contient des informations sur 1290 établissements hospitaliers. Ces données sont basées sur le jeu de données original du National Health Service (NHS) britannique, mais ont été modifiées et enrichies pour les besoins de la démonstration.

Ajout de spécialités médicales : Une liste de spécialités a été générée à partir des informations du NHS, puis attribuée de manière aléatoire à chaque hôpital. (cf document : Données de référence sur les spécialités NHS)
Capacité d'accueil : Un nombre aléatoire de lits disponibles, compris entre 0 et 150, a été assigné à chaque hôpital.

Il est important de noter que ce **jeu de données a été créé uniquement à des fins de démonstration et de développement**. Les informations qu'il contient ne reflètent

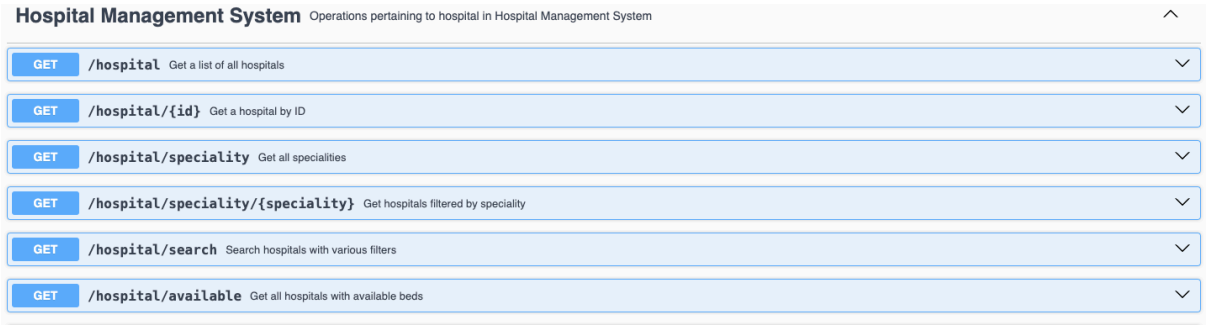
pas la réalité des établissements hospitaliers en termes de capacité, de spécialités offertes ou de sécurité des données. Ce dataset ne doit en aucun cas être utilisé pour des décisions opérationnelles ou stratégiques dans un contexte réel.

Cette approche permet de travailler avec un **volume de données représentatif** tout en préservant la confidentialité des informations sensibles du système de santé.

Développement des API REST

Création des endpoints

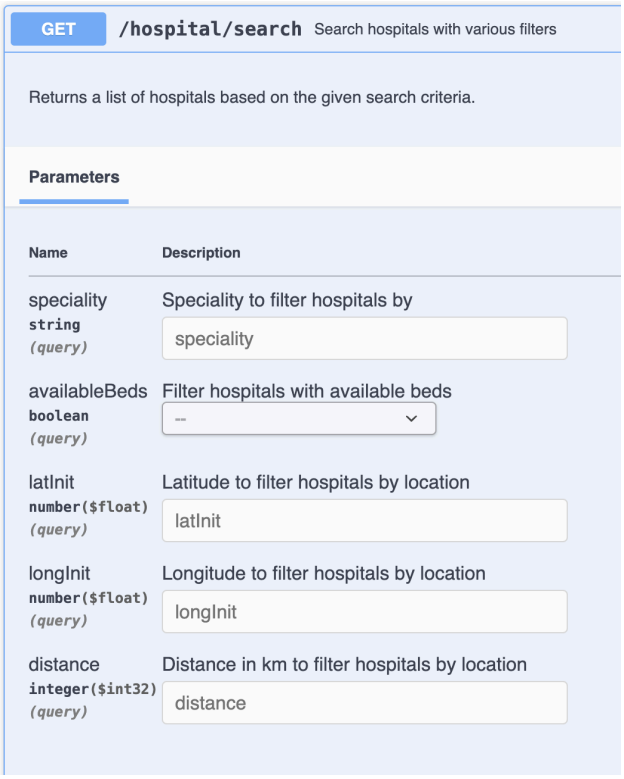
Les endpoints nécessaires pour les opérations de recherche des hôpitaux ont été implémentés.



Documentation Swagger: Liste des routes

- récupération de tous les hôpitaux, ou d'un hôpital précis en passant l'identifiant en paramètre
- récupération de toutes les spécialités présentes en base de données
- récupération de tous les hôpitaux avec une spécialité demandé dans la requête
- récupération de tous les hôpitaux avec des lits disponibles

Le endpoint qui nous intéresse le plus est le "/hospital/search". Il s'agit d'une route acceptant plusieurs paramètres passés dans le corps de la requête.



Documentation Swagger: Paramètres de la route /hospital/search

Tous les paramètres sont optionnels, cette route peut donc être utilisée avec plusieurs combinaisons. Les lits disponibles, la spécialité et la localisation.

Concernant les données de localisation (“latInit”, “longInit”), elles sont récupérées dans le front-end grâce à une API externe “https://api.opencagedata.com” avec comme paramètre la localisation donnée par l’utilisateur dans le formulaire. Les coordonnées sont extraites pour pouvoir être transmises à l’API via la route “/hopistal/search”

Documentation de l’API

Les endpoints sont documentés avec Swagger sur une URL locale au lancement du projet. Les informations pour y accéder sont présentes dans le README, partie “Documentation”.

Chaque endpoint a été documenté, décrivant son utilisation, les paramètres requis, facilitant ainsi les tests et les futures intégrations.

Une personnalisation plus approfondie de la documentation Swagger est à envisager pour assurer une intégration du service avec d’autres systèmes. Notamment ajouter une documentation pour les retours d’erreurs possibles, ainsi qu’un exemple de réponse attendu en cas de réussite de la requête.

Tests

Tests de charges

L’objectif est d’évaluer la capacité du système à gérer un volume élevé de trafic et identifier ses limites de performance.

Pour faire ces tests l’outil utilisé est **JMeter**, avec une configuration de 2000 utilisateurs simulés, ce qui représente plus du double du minimum exigé pour cette PoC. Le but est d’assurer une performance idéale pour une utilisation extra-ordinaire du service avec ces 2000 utilisateurs simultanés.
Le but est d’identifier les points de rupture du système, les temps de réponse sous charge maximale, et les éventuelles erreurs ou échecs.

Tests fonctionnels

L’objectif est de vérifier que chaque endpoint de l'application répond correctement et respecte les spécifications fonctionnelles.

Pour mettre en place ces tests un script “.http” a été utilisé avec l’IDE IntelliJ IDEA.
Chaque requête dans le fichier .http représente une fonctionnalité ou un cas d'utilisation à vérifier.
Les réponses doivent contenir les données correctes, avoir les bons statuts HTTP (200, 404, 500, etc.), et répondre dans un délai acceptable.
Ces tests vont permettre de définir les améliorations nécessaires, notamment sur les fonctionnalités de recherches plus avancées.

Tests unitaires

Les tests unitaires sont des composants essentiels pour garantir la qualité et la fiabilité du code. Ils permettent de vérifier que chaque unité de code fonctionne comme prévu. Pour le service des hôpitaux, nous avons mis en place une série de tests unitaires couvrant les différents aspects des fonctionnalités de recherche et de récupération des hôpitaux.

Les tests unitaires ont été développés en utilisant JUnit et Mockito, et incluent les cas suivants :

- **testGetAllHospital()** : Vérifie que la méthode getAllHospital() renvoie la liste complète des hôpitaux présents dans le dépôt.

- **testSearchHospitalsWithResults()** : Vérifie que la méthode searchHospitals() renvoie les résultats corrects lorsqu'une spécialité spécifique est recherchée.
- **testSearchHospitalsNoResults()** : Vérifie que la méthode searchHospitals() renvoie une liste vide lorsque aucune correspondance n'est trouvée pour la spécialité recherchée.
- **testSearchHospitalsWithAvailableBeds()** : Vérifie que la méthode searchHospitals() renvoie les hôpitaux avec des lits disponibles.
- **testSearchHospitalsWithLocation()** : Vérifie que la méthode searchHospitals() renvoie les hôpitaux situés dans un périmètre défini par une latitude et une longitude.
- **testSearchHospitalsWithSpecialityAndLocation()** : Vérifie que la méthode searchHospitals() renvoie les hôpitaux correspondant à une spécialité et situés dans un périmètre spécifique.

Les résultats de ces tests unitaires montrent que chaque méthode du service fonctionne comme attendu dans des conditions de test contrôlées. Les tests sont exécutés avant chaque nouvelle version pour garantir que les modifications du code n'interdisent pas de régressions.

Tests d’intégration

Les tests d'intégration visent à vérifier que les différents composants du système fonctionnent correctement ensemble. Ces tests sont cruciaux pour détecter des problèmes d'interaction entre les différentes parties du code, qui ne seraient pas visibles lors des tests unitaires.

Pour le service des hôpitaux, les tests d'intégration ont été réalisés en utilisant un environnement de test Spring Boot, avec une base de données H2 en mémoire. Les tests suivants ont été effectués :

- **testGetAllHospital()** : Vérifie que l'intégration entre le service et le dépôt fonctionne correctement en renvoyant tous les hôpitaux de la base de données.
- **testSearchHospitalsWithResults()** : Vérifie que la méthode searchHospitals() peut correctement interroger la base de données pour des spécialités spécifiques.
- **testSearchHospitalsNoResults()** : Vérifie que la méthode searchHospitals() renvoie une liste vide lorsque aucune correspondance n'est trouvée.
- **testSearchHospitalsWithAvailableBeds()** : Vérifie que la méthode searchHospitals() renvoie les hôpitaux avec des lits disponibles en interrogeant la base de données.
- **testSearchHospitalsWithLocation()** : Vérifie que la méthode searchHospitals() peut interroger la base de données pour des hôpitaux dans un périmètre défini par des coordonnées géographiques.
- **testSearchHospitalsWithSpecialityAndLocation()** : Vérifie que la méthode searchHospitals() peut combiner la recherche par spécialité et par localisation.

- **testSearchHospitalsWithMultipleCriteria()** : Vérifie que la méthode searchHospitals() peut combiner plusieurs critères de recherche (spécialité, lits disponibles, localisation).

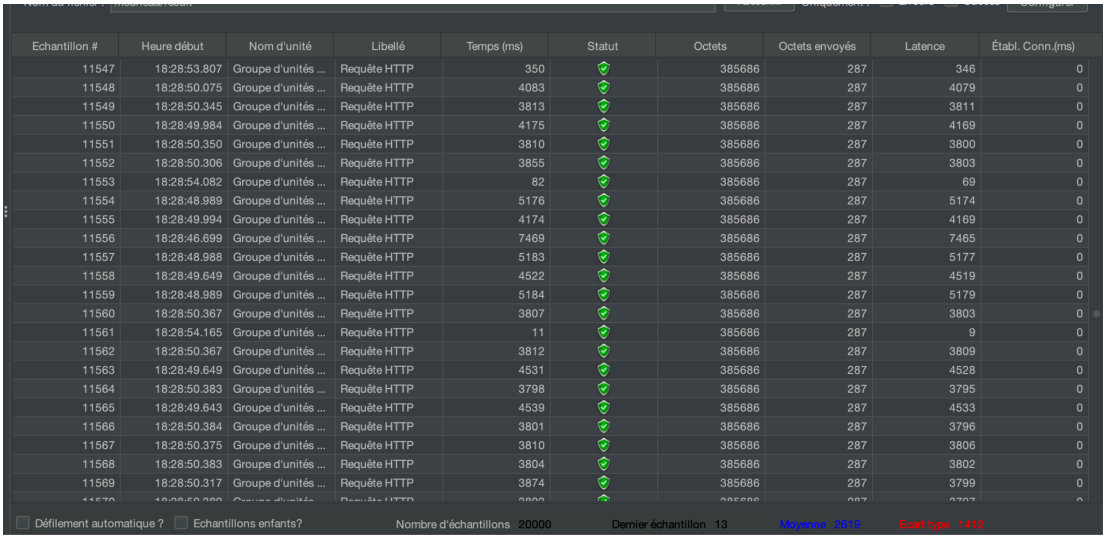
4. Résultats

Performances des API

Les tests de performance des API ont révélé que, sous des conditions normales d'utilisation, les temps de réponse restent majoritairement en dessous de 200 ms, conformément aux objectifs initiaux. Cependant, sous des charges élevées, une augmentation significative a été observée, avec des temps de réponse moyens atteignant 2619 ms. Cette saturation indique un besoin d'optimisation, notamment pour garantir une performance constante et réduire la variabilité observée.
Les tests de charge ont été effectués avec **JMeter**, simulant jusqu'à 2000 utilisateurs, révélant que le système peut supporter des charges élevées mais présente des variabilités importantes dans les temps de réponse, suggérant une saturation du système.
Dans le cadre des 800 requêtes par seconde demandées pour ces tests, ces résultats sont plus que corrects pour un cas de 2000 requêtes. Ce qui assure une bonne performance du système.

Robustesse du Système

La moyenne de 2619 ms est relativement élevée. Cela peut indiquer que le système commence à montrer des signes de saturation sous une charge plus élevée. L'écart type élevé montre une grande variabilité dans les temps de réponse, ce qui n'est pas idéal pour l'expérience utilisateur. Le fait que toutes les requêtes réussissent est un point positif, montrant que le système est stable malgré des temps de réponse plus longs.



Echantillon #	Heure début	Nom d'unité	Libellé	Temps (ms)	Statut	Octets	Octets envoyés	Latence	Établ. Conn.(ms)
11547	18:28:53.807	Groupe d'unités ...	Requête HTTP	350		385686	287	346	0
11548	18:28:50.075	Groupe d'unités ...	Requête HTTP	4083		385686	287	4079	0
11549	18:28:50.345	Groupe d'unités ...	Requête HTTP	3813		385686	287	3811	0
11550	18:28:49.984	Groupe d'unités ...	Requête HTTP	4175		385686	287	4169	0
11551	18:28:50.350	Groupe d'unités ...	Requête HTTP	3810		385686	287	3800	0
11552	18:28:50.306	Groupe d'unités ...	Requête HTTP	3855		385686	287	3803	0
11553	18:28:54.082	Groupe d'unités ...	Requête HTTP	82		385686	287	69	0
11554	18:28:48.989	Groupe d'unités ...	Requête HTTP	5176		385686	287	5174	0
11555	18:28:49.994	Groupe d'unités ...	Requête HTTP	4174		385686	287	4169	0
11556	18:28:46.699	Groupe d'unités ...	Requête HTTP	7469		385686	287	7465	0
11557	18:28:48.988	Groupe d'unités ...	Requête HTTP	5183		385686	287	5177	0
11558	18:28:49.649	Groupe d'unités ...	Requête HTTP	4522		385686	287	4519	0
11559	18:28:48.989	Groupe d'unités ...	Requête HTTP	5184		385686	287	5179	0
11560	18:28:50.367	Groupe d'unités ...	Requête HTTP	3807		385686	287	3803	0
11561	18:28:54.165	Groupe d'unités ...	Requête HTTP	11		385686	287	9	0
11562	18:28:50.367	Groupe d'unités ...	Requête HTTP	3812		385686	287	3809	0
11563	18:28:49.649	Groupe d'unités ...	Requête HTTP	4531		385686	287	4528	0
11564	18:28:50.383	Groupe d'unités ...	Requête HTTP	3798		385686	287	3795	0
11565	18:28:49.643	Groupe d'unités ...	Requête HTTP	4539		385686	287	4533	0
11566	18:28:50.384	Groupe d'unités ...	Requête HTTP	3801		385686	287	3796	0
11567	18:28:50.375	Groupe d'unités ...	Requête HTTP	3810		385686	287	3806	0
11568	18:28:50.383	Groupe d'unités ...	Requête HTTP	3804		385686	287	3802	0
11569	18:28:50.317	Groupe d'unités ...	Requête HTTP	3874		385686	287	3799	0

Défilement automatique ? Echantillons enfants? Nombre d'échantillons 20000 Dernier échantillon 13 Moyenne 3819 Ecart type 1412

Tableau de résultats JMeter

Sécurité

Dans le cadre de cette PoC la mise en place de sécurité avancée n'a pas été réalisée. Certaines mesures de base ont été mises en place grâce à l'utilisation du framework Spring Boot. Les r**equêtes préparées** sont utilisées par défaut, ce qui offre une protection contre les attaques par injection SQL. De plus, **Spring Security**, inclus dans Spring Boot, fournit une protection **CSRF** (Cross-Site Request Forgery) de base. Ces mesures, bien que limitées, constituent un point de départ pour la sécurisation du système. Des recommandations supplémentaires seront introduites dans l'Analyse des Résultats

Tests Fonctionnels

Les tests fonctionnels ont couvert chaque endpoint, vérifiant leur conformité aux spécifications fonctionnelles. Les résultats ont montré que tous les endpoints répondaient correctement aux attentes, avec des réponses adéquates et des statuts HTTP appropriés. Par exemple, le endpoint GET /hospital a un temps de réponse de 45 ms, tandis que GET /hospital/1 répond en 10 ms.

Des requêtes plus complexes, comme GET /hospital/search?speciality=cardiology&availableBeds=true, prennent environ 14 ms, ce qui est acceptable pour ce type de recherche combinée. Le temps de réponse global pour l'ensemble des requêtes a été en moyenne de 256 ms, ce qui confirme la bonne performance des endpoints sous des conditions normales.

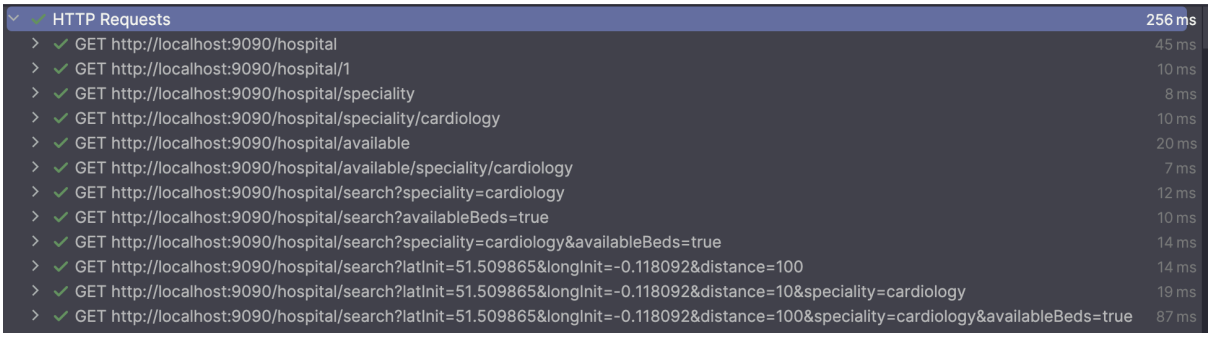
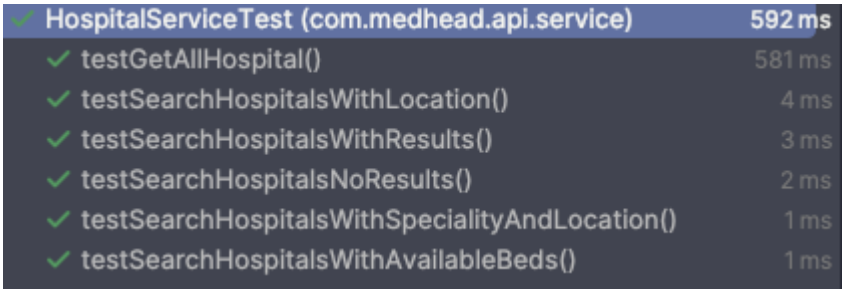


Tableau de réponse du script .http: route et temps de réponses par requêtes

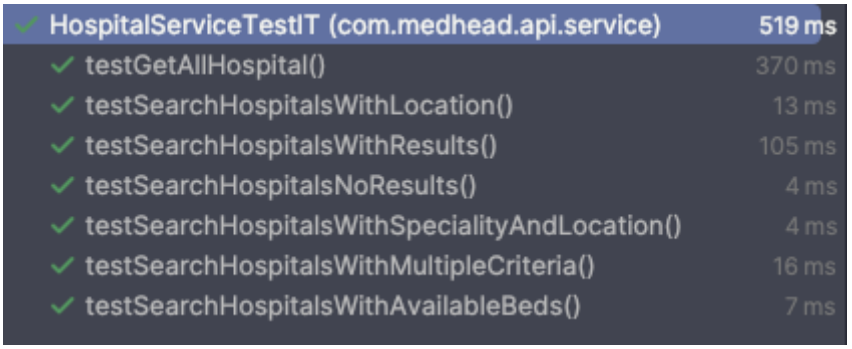
Tests Unitaires et d'Intégration

Les tests unitaires ont couvert les principales méthodes et fonctionnalités du service des hôpitaux. Les résultats de ces tests montrent que chaque méthode fonctionne comme prévu dans des conditions contrôlées. Par exemple, la méthode testGetAllHospital() a une durée de réponse de 547 ms, et la méthode testSearchHospitalsWithLocation() prend seulement 3 ms, démontrant l'efficacité du code. Les tests plus complexes, comme testSearchHospitalsWithMultipleCriteria(), affichent des temps de réponse de 16 ms, ce qui est acceptable pour ce type de requête.



Résultats des tests du fichier “HospitalServiceTest” contenant les tests unitaires

En complément, les tests d'intégration ont confirmé que les différents composants du système fonctionnent correctement ensemble, sans problèmes majeurs d'interaction détectés. La méthode testGetAllHospital(), par exemple, a montré une intégration fluide avec un temps de réponse de 370 ms. Les résultats des tests, tels que ceux du fichier “HospitalServiceTestIT”, illustrent que les interactions complexes, comme la recherche de lits disponibles (testSearchHospitalsWithAvailableBeds()), restent performantes avec des temps de réponse allant jusqu'à 7 ms en intégration.



Résultats des tests du fichier "HospitalServiceTestIT" contenant les tests d'intégration

Ces résultats positifs des tests unitaires et d'intégration indiquent que le système est bien conçu et prêt pour des mises en œuvre plus larges. Les performances en termes de temps de réponse sont globalement satisfaisantes, et l'intégrité des interactions entre les composants est assurée.

5. Analyse des Résultats

Analyse des performances et de la robustesse

Les temps de réponse en conditions normales sont satisfaisants, mais des optimisations sont nécessaires sous charge pour améliorer la performance et réduire la variabilité des résultats.
Les tests de charge ont révélé que le système **atteint ses limites de performance à des charges élevées**(2000 utilisateurs), suggérant la nécessité d'optimisations au niveau du traitement des requêtes et de la gestion des ressources.
Une piste d'amélioration serait l'implémentation de **mécanismes de mise en cache** pour les requêtes fréquentes, réduisant ainsi la charge sur le serveur. L**'optimisation des requêtes SQL** et l'implémentation de la **pagination** pourraient également contribuer à améliorer les performances.

Pour stabiliser les performances sous charge, il est recommandé d'utiliser des techniques avancées comme le **partitionnement de la base de données** (sharding), qui permet de répartir la charge de travail et de réduire les temps de réponse.
De plus, la mise en place d'un **système de surveillance proactive**, tel que Spring Boot Actuator, aiderait à identifier et résoudre les goulets d'étranglement en temps réel, assurant ainsi une expérience utilisateur cohérente. Ces mécanismes permettent non seulement de gérer les ressources de manière plus efficace, mais aussi d'améliorer la robustesse du système face à des charges élevées.

Intégration

Le projet a été conçu avec comme objectif de s'intégrer dans une architecture de **microservices**, ce qui permet une **flexibilité** et une **scalabilité accrues**. Chaque composant du système peut **fonctionner de manière indépendante** tout en communiquant efficacement avec les autres services. Cette approche modulaire facilite l'intégration de nouveaux services et la mise à jour des services existants sans perturber l'ensemble du système. Par exemple, il est possible d'intégrer facilement un service externe pour la prise en charge d'un patient ou pour la réservation d'un lit d'hôpital via le front-end ou le back-end.

Le respect du **format MVC** (Modèle-Vue-Contrôleur) dans la conception du système renforce encore cette flexibilité et cette robustesse.
En séparant les préoccupations, le format MVC permet une gestion claire et distincte des données (Modèle), de l'interface utilisateur (Vue) et de la logique d'application (Contrôleur). Cette séparation facilite non seulement le développement et la maintenance, mais aussi l'intégration de nouveaux composants ou services, puisque chaque couche peut être modifiée indépendamment des autres.

Pour améliorer encore l'intégration, il serait bénéfique de **standardiser les formats de données et les protocoles de communication** utilisés dans les échanges entre systèmes. L'adoption de standards ouverts, tels que **JSON** pour les formats de données et **REST** pour les protocoles de communication, peut également faciliter l'intégration future et réduire les efforts nécessaires pour maintenir la compatibilité. La mise en place de **documentation** des systèmes, de **flux de données**(XML) ou de webhooks permettrait de simplifier la communication et le respect des contrats entre les systèmes.

L'utilisation de services d'orchestration peut également aider à gérer les interactions entre les microservices, assurant ainsi une coordination efficace et une gestion optimale des workflows complexes. Par exemple, des outils comme Kubernetes peuvent être utilisés pour orchestrer les conteneurs de microservices, en garantissant qu'ils sont correctement déployés, surveillés et mis à l'échelle en fonction de la demande.

En résumé, la conception basée sur des microservices et le respect du format MVC répondent non seulement aux besoins actuels mais offrent également une base solide pour l'ajout de nouvelles fonctionnalités et l'intégration de services supplémentaires, garantissant ainsi une évolutivité et une adaptabilité continues.

Sécurité

L'analyse de la sécurité révèle que des améliorations significatives sont nécessaires pour rendre le système pleinement sécurisé et conforme aux normes de protection des données de santé. Bien que les protections de base offertes par Spring Boot soient en place, plusieurs pistes d'amélioration sont à considérer :

1. **Protection des données sensibles** : Le chiffrement des données au repos et en transit est recommandé, particulièrement pour les informations médicales. Cette mesure nécessite l'utilisation d'algorithmes de chiffrement robustes comme AES-256 pour les données stockées et TLS 1.3 pour les communications réseau.

2. **Authentification et autorisation** : Un système d'authentification multi-facteurs (par exemple, mot de passe + code SMS) peut être adopté, accompagné d'une gestion fine des autorisations basée sur les rôles pour contrôler l'accès aux différentes parties du système, notamment la base de données.

3. **Sécurisation des API** : L'authentification par token JWT avec une durée de validité limitée et une rotation fréquente des clés est recommandée. La limitation du taux de requêtes (rate limiting) doit être configurée pour protéger contre les attaques par déni de service, avec des seuils adaptés selon le type d'utilisateur et d'opération.

4. **Gestion sécurisée des logs** : Un système de logging sécurisé est conseillé pour tracer les activités sensibles sans exposer d'informations confidentielles. Ce système doit inclure la anonymisation des données personnelles dans les logs, le chiffrement des fichiers de logs, et l'établissement d'une politique de rétention et de rotation des logs.

5. **Formation** : Les compétences en sécurité des développeurs et de leurs collaborateurs sont d'une importance capitale. Un programme de formation rigoureux et continu doit être établi, couvrant les meilleures pratiques de sécurité, les menaces émergentes et les techniques de protection spécifiques au domaine de la santé. Des formations régulières sur les nouvelles vulnérabilités, les techniques de codage sécurisé, et les réglementations spécifiques au secteur médical doivent être dispensées si nécessaire.

6. **Audits de sécurité réguliers** : Des audits internes et externes doivent être planifiés régulièrement pour identifier et corriger les vulnérabilités potentielles. Ces audits devraient inclure des tests d'intrusion, des analyses de code statique et dynamique, et des revues de configuration des systèmes.

7. **Conformité réglementaire** : La conformité avec les réglementations spécifiques au secteur de la santé, telles que le RGPD, doit être assurée. Cela implique l'établissement de processus de gestion des données personnelles, de notification des violations, et de documentation des mesures de sécurité.

Fonctionnalités

Les résultats des tests fonctionnels confirment que les endpoints répondent aux spécifications, mais des améliorations sont nécessaires pour certaines fonctionnalités avancées. Par exemple, le endpoint GET /hospital a montré un temps de réponse moyen de 45 ms, ce qui est satisfaisant pour une requête simple. Les requêtes plus complexes, telles que GET /hospital/search?speciality=cardiology&availableBeds=true, affichant un temps de réponse de 14 ms, ce qui est bien pour la complexité de la tâche. Cependant, des optimisations supplémentaires pourraient être envisagées pour maintenir ces performances sous des charges plus élevées.

Certaines fonctionnalités avancées, telles que la recherche multi-critères, ont montré des temps de réponse légèrement plus élevés. Le endpoint GET /hospital/search qui permet de combiner plusieurs critères de recherche (spécialité, lits disponibles, localisation) a affiché un temps de réponse de 16 ms. Bien que ces temps de réponse soient acceptables, il est essentiel de continuer à optimiser ces endpoints pour garantir des performances accrues, surtout lorsque le système est soumis à une utilisation intensive. Pour ces optimisations, il pourrait être utile d'examiner la manière dont les critères de recherche sont combinés et traités, en optimisant les algorithmes de recherche et en utilisant des techniques telles que l'indexation avancée ou les filtres en mémoire.

Qualité du code et maintenabilité

Plusieurs points sont à aborder concernant la qualité du code, la maintenabilité et les axes d’améliorations à envisager.

- **Couverture des tests**
  Les tests unitaires et d'intégration couvrent les principales méthodes et fonctionnalités, assurant que chaque composant fonctionne comme prévu. Une couverture de test plus élevée est à prévoir dans le cas de l’adoption de cette architecture, cela permettra de mieux détecter et corriger les bugs rapidement, augmentant ainsi la fiabilité du système. Pour pousser la couverture des tests, la mise en place de **développement piloté par les tests**(TDD) est recommandée.

- **Respect du modèle MVC**
  Le respect du modèle **MVC** (Modèle-Vue-Contrôleur) assure une séparation claire des préoccupations. Cela facilite le développement, la maintenance et l'intégration de nouveaux composants ou services, rendant le code plus modulaire et extensible. Le modèle MVC est assez répandu de nos jours et ne nécessite pas de formations étendues pour le mettre en place.

- **Bonnes pratiques de développement**
  Le code suit les bonnes pratiques de développement, incluant des conventions de nommage claires et une documentation appropriée. Ces pratiques améliorent la lisibilité et la maintenabilité du code.

- **Normes HTTPS**
  L'utilisation des normes **HTTPS** pour les communications doit être mise en place pour garantir la sécurité des données échangées. Cela est primordial pour protéger les informations sensibles des patients et maintenir la conformité avec les régulations appliquées par les différentes lois liées à la santé et aux systèmes informatiques.

- **Documentation et Communication**
  Chaque endpoint est documenté avec **Swagger**, ce qui facilite la compréhension et l'utilisation des API. La documentation doit être mise à jour régulièrement, pour chaque release pouvant impacter son contenu.
  Une documentation claire et à jour aide les développeurs à prendre en main et à maintenir le code plus facilement.

- **Maintenabilité**
  Pour maintenir le code efficacement, il est recommandé d'adopter des pratiques de **refactoring régulier**, d'utiliser des design patterns appropriés et d'automatiser les tests et les déploiements avec des outils d'**intégration continue** (CI) et de **déploiement continu** (CD). Ces techniques assurent que le code reste propre, structuré et facile à gérer.

- **Monitoring**
  La surveillance des systèmes est un point crucial dans l'amélioration continue des systèmes, car elle permet d'identifier rapidement les problèmes, d'analyser les performances et de garantir la disponibilité du service.
  Elle peut être mise en place par divers outils en fonction des besoins spécifiques du projet.
  L'application **Sentry** est **recommandée** pour le **suivi des erreurs et des performances**. Sentry offre une visibilité en temps réel sur les erreurs et les exceptions, facilitant ainsi la détection et la correction rapide des bugs. En surveillant les performances, Sentry aide également à identifier les goulets d'étranglement et à optimiser les ressources, garantissant ainsi une expérience utilisateur fluide et fiable.
  L'utilisation de tels outils permet non seulement de maintenir la qualité du service, mais aussi d'améliorer continuellement l'efficacité et la résilience du système.

6. Conclusion

La proof of concept (PoC) pour le système d'intervention d'urgence de MedHead a démontré que l'architecture **microservice proposée est capable de répondre efficacement aux besoins des institutions médicales du consortium.**

Les principaux objectifs de la PoC ont été atteints. Les temps de réponse des API sous des conditions normales sont majoritairement en dessous de 200 ms, ce qui est conforme aux attentes. Cependant, sous des charges élevées, le système a montré des signes de saturation avec des temps de réponse moyens nécessitant des optimisations supplémentaires.

Tous les endpoints API ont fonctionné correctement, retournant les réponses attendues avec les statuts HTTP appropriés.
Les tests fonctionnels ont révélé des **performances acceptables**, même pour des requêtes complexes combinant plusieurs critères.
Le système a été conçu de manière modulaire en suivant une architecture de **microservices**, facilitant ainsi l'intégration avec les systèmes existants et la possibilité d'extensions futures sans perturbations majeures.

Les mécanismes de sécurité mis en place sont **basiques**, et des **efforts supplémentaires sont nécessaires** pour renforcer la protection des données sensibles des patients et des hôpitaux.

7. Recommandations

Il est recommandé d'améliorer la gestion des requêtes et des ressources pour réduire la variabilité des temps de réponse sous charge, de mettre en place des mécanismes de mise en cache pour les requêtes fréquentes et d'optimiser les requêtes SQL. En matière de sécurité, il est crucial d'intégrer des mesures de sécurité avancées, et de mettre en place une surveillance proactive..

L'amélioration de la documentation des API, y compris des exemples de réponses et une gestion des erreurs détaillée, ainsi que la formation des utilisateurs finaux et des administrateurs système, est essentielle pour garantir une adoption fluide du nouveau système. Enfin, augmenter la couverture des tests unitaires et d'intégration, adopter des pratiques de développement pilotées par les tests (TDD) et utiliser des outils d'intégration continue (CI) et de déploiement continu (CD) sont fortement recommandés pour assurer un code propre et structuré.

En conclusion, la PoC a validé que l'architecture proposée est adéquate pour les besoins de MedHead, tout en identifiant des axes d'amélioration importants pour garantir une performance optimale et une sécurité renforcée.