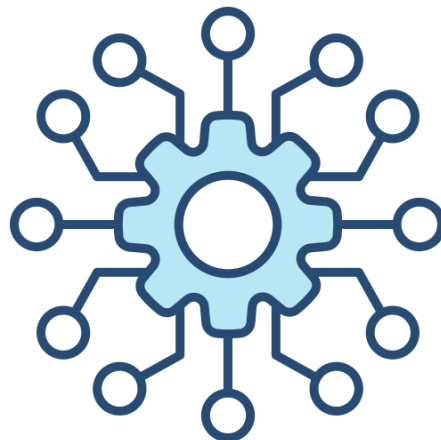




Rapport d'évaluation du POC

Système d'intervention d'urgence

**Proof of Concept pour la création d'une
Plateforme de Gestion des Réservations de Lits
d'Hôpitaux en Situation d'Urgence**



ZANTOUR Wael

Mastère 2 Expert en Ingénierie Logicielle

2023-2024

Objet du document :

Ce document détaille les principes architecturaux suivis, les choix technologiques effectués, ainsi que les étapes de développement et d'implémentation.

Il décrit également les résultats obtenus et les conclusions tirées de ce PoC, en vue de la mise en œuvre d'une solution complète et déployable à plus grande échelle.

Historique des révisions :

Numéro de version	Auteur	Description	Date de modification
1.0	ZANTOUR Wael	Livraison initiale	20/08/2024

1. Introduction	5
1.1. Contexte	5
1.2. Objectifs du PoC	5
2. Description du Projet	6
2.1. Enjeux et Problématique :	6
2.2. Stack Technique Utilisée	6
Front-end	6
Backend	7
Docker	7
Gestion du Versionnement	8
Sécurité	8
2.3. Vision de la Conception de l'Architecture	9
2.4. Intégration de Kafka	11
3. Mise en Œuvre des Microservices	12
3.1. Présentation Générale	12
3.2. Détails des Microservices	12
3.2.1. User Authentication Microservice	12
3.2.2. Hospital Management Microservice	13
3.2.3. Microservice de Réservation	14
3.2.4. Notification Microservice	15
3.2.5. API Gateway	16
3.2.6. Discovery Server (Eureka)	16
3.2.7. Config Server	17
3.3. Orchestration et Déploiement avec Docker Compose	18
4. Tests et Validation	19
4.1. Introduction aux Stratégies de Test	19
4.2. Tests Unitaires et d'Intégration	20
4.3. Tests API avec Postman	21
4.4. Tests de Montée en Charge et de performance	24
4.5. Tests End-to-End (E2E)	26
5. Pipelines CI/CD avec GitHub Actions	27
5.1. Introduction	27
5.2. Pipeline CI pour le Backend	28
5.3. Pipeline CI pour le Frontend	28
5.4. Avantages de l'Implémentation des Pipelines CI	29
6. Illustrations de l'Interface Utilisateur et de la Documentation API	31
6.1. Accès en tant qu'Administrateur et Patient	31

MedHead Consortium

Accès en tant qu'Administrateur	31
Accès en tant que Patient	31
6.2. Interface Utilisateur Angular	32
6.3. Documentation API Swagger	37
7. Conclusion et Recommandations	38
7.1. Évaluation de la Conformité à la Demande	38
7.2. Recommandations pour la Mise en Œuvre de la Solution Finale	39
7.3. Conclusion	40

1. Introduction

1.1. Contexte

Le projet de plateforme de gestion des réservations de lits d'hôpitaux est initié par un consortium regroupant plusieurs institutions médicales. Actuellement, les organisations membres de ce consortium utilisent une diversité de technologies et de systèmes informatiques, ce qui complique la coordination des soins, particulièrement dans les situations d'urgence où la rapidité d'intervention est cruciale. Ces institutions souhaitent donc une solution unifiée qui permettrait de gérer de manière centralisée les recommandations et les réservations de lits d'hôpitaux en cas d'urgence, tout en respectant les différentes technologies existantes au sein des différentes entités.

1.2. Objectifs du PoC

Le principal objectif de ce Proof of Concept (PoC) est de démontrer la faisabilité technique de l'architecture envisagée pour la plateforme, en se concentrant sur des fonctionnalités clés telles que la gestion des utilisateurs, la recherche des hôpitaux, la réservation de lits d'hôpitaux, et l'envoi de notifications de confirmation. Ce PoC vise à valider que l'approche microservices, combinée à des technologies telles que Spring Boot pour le backend et Angular pour le frontend, répond efficacement aux exigences du consortium.

Plus précisément, le PoC vise à :

- **Démontrer la viabilité de l'architecture microservices** : Tester si une architecture décentralisée peut efficacement gérer des opérations critiques telles que l'authentification des utilisateurs, la recherche d'hôpitaux en fonction de la spécialité et de la disponibilité des lits, et la réservation des lits d'hôpitaux.
- **Valider l'intégration des services via Kafka** : Utiliser Kafka pour permettre une communication asynchrone entre les différents microservices, notamment pour l'envoi de notifications de confirmation de réservation, garantissant ainsi la résilience du système face aux pics de charge et aux interruptions temporaires de services.
- **Assurer la sécurité des données** : Mettre en place une authentification sécurisée par JWT (JSON Web Token) pour les utilisateurs, en veillant à ce que seules les personnes autorisées puissent accéder aux fonctionnalités critiques de la plateforme.

Ce PoC servira de base pour évaluer les performances, la sécurité, et la convivialité de la solution, avant de procéder à une éventuelle mise en production à grande échelle dans les différentes institutions membres du consortium.

2. Description du Projet

2.1. Enjeux et Problématique :

Le consortium d'institutions médicales, initiateur de ce projet, est confronté à une diversité de technologies et de systèmes qui rendent difficile la coordination des soins, en particulier dans les situations d'urgence où la rapidité et la précision sont essentielles. L'objectif est de développer une plateforme unifiée capable de gérer efficacement les processus critiques de recherche et de réservation de lits d'hôpitaux, tout en garantissant la sécurité des données des patients et la résilience du système face aux interruptions.

2.2. Stack Technique Utilisée

Front-end

- **Angular** : Angular est utilisé pour développer l'interface utilisateur, qui est à la fois réactive et intuitive. L'architecture modulaire d'Angular facilite la gestion de l'état de l'application, ainsi que l'intégration avec les services backend.
- **TypeScript** : Choisi pour son typage statique, TypeScript améliore la qualité du code en détectant les erreurs dès la compilation, ce qui contribue à un développement plus sûr et plus efficace.
- **OpenAPI** : OpenAPI est utilisé dans le frontend pour générer automatiquement les services HTTP qui interagissent avec les API REST exposées par les microservices backend. Grâce à la spécification OpenAPI, les services Angular sont générés de manière cohérente avec les endpoints définis, réduisant ainsi les erreurs humaines et facilitant l'intégration continue entre le frontend et le backend.
- **Cypress** : Framework de test end-to-end, utilisé pour valider les flux utilisateur en simulant des interactions réelles dans le navigateur. Cypress permet de s'assurer que toutes les fonctionnalités critiques fonctionnent correctement, du login à la réservation de lits des hôpitaux.

Backend

Afin de satisfaire aux exigences, les services back-end (micro-services) ont été construits sur une base Java (v 17).

- **Spring Boot** : Framework principal pour le développement des microservices. Spring Boot simplifie la configuration et le déploiement des applications Java, tout en offrant une intégration native avec d'autres composants de l'écosystème Spring (Spring Security, Spring Data, Lombok, etc.).
- **Spring Security** : Utilisé pour gérer l'authentification et l'autorisation des utilisateurs, garantissant que seules les personnes autorisées peuvent accéder aux services critiques via JWT.
- **Eureka (Netflix OSS)** : Service de découverte permettant aux microservices de s'enregistrer dynamiquement et de découvrir les autres services sans configuration statique.
- **Kafka** : Plateforme de messagerie distribuée utilisée pour la communication asynchrone entre les microservices, particulièrement pour l'envoi de notifications.
- **PostgreSQL** : Base de données relationnelle utilisée pour stocker les données critiques de la plateforme, telles que les utilisateurs et les réservations.
- **MongoDB** : Base de données NoSQL utilisée pour stocker les données non structurées.
- **Swagger/OpenAPI** : Utilisé pour documenter et exposer les API REST des microservices. Swagger UI permet aux développeurs de tester les API directement depuis le navigateur, tandis que les schémas OpenAPI assurent que la documentation est toujours à jour et fidèle au code.

Docker

- **Docker Compose** : Docker Compose est utilisé pour orchestrer les conteneurs de chaque microservice. Le fichier `docker-compose.yml` définit les services, leurs dépendances, et les réseaux nécessaires pour que tous les composants fonctionnent ensemble de manière cohérente.

Gestion du Versionnement

Git et GitHub : Le versionnement du code source est géré à l'aide de Git, avec GitHub comme plateforme d'hébergement des dépôts. Le projet utilise un modèle de branches Git pour structurer le développement, avec les branches suivantes :

- **Branche `main` :** C'est la branche principale du projet, contenant la version stable et prête pour la production. Les fonctionnalités complètement testées et validées sont fusionnées dans cette branche après avoir été approuvées via une pull request.
- **Branche `develop` :** Cette branche sert de base pour le développement actif. C'est ici que les nouvelles fonctionnalités et améliorations sont intégrées avant d'être stabilisées pour une prochaine version. Une fois que les changements sont suffisamment stables, ils sont fusionnés dans la branche `main`.
- **Branche `hotfix_branch` :** Utilisée pour appliquer des corrections urgentes à la version en production. Si un problème critique est détecté, une branche `hotfix` est créée à partir de `main` pour corriger le bug. Après la correction, la branche `hotfix` est fusionnée dans `main` et `develop` pour s'assurer que la correction est propagée à la version en développement.
- **Feature Branches `feature_branch` :** Chaque nouvelle fonctionnalité est développée dans une branche distincte, dérivée de la branche principale.
- **Branche `ci/pipeline` :** Cette branche est utilisée pour configurer et tester les pipelines CI/CD. Elle permet de s'assurer que le pipeline d'intégration continue fonctionne correctement avant de l'intégrer aux branches principales du projet.

Sécurité

La sécurité est cruciale pour cette plateforme, qui traite des données médicales sensibles. Plusieurs mesures ont été mises en place pour assurer une protection adéquate :

- **Authentication JWT :** Chaque utilisateur authentifié reçoit un token JWT pour sécuriser les communications entre le frontend et les microservices backend, garantissant l'accès aux services protégés en fonction des autorisations de l'utilisateur.

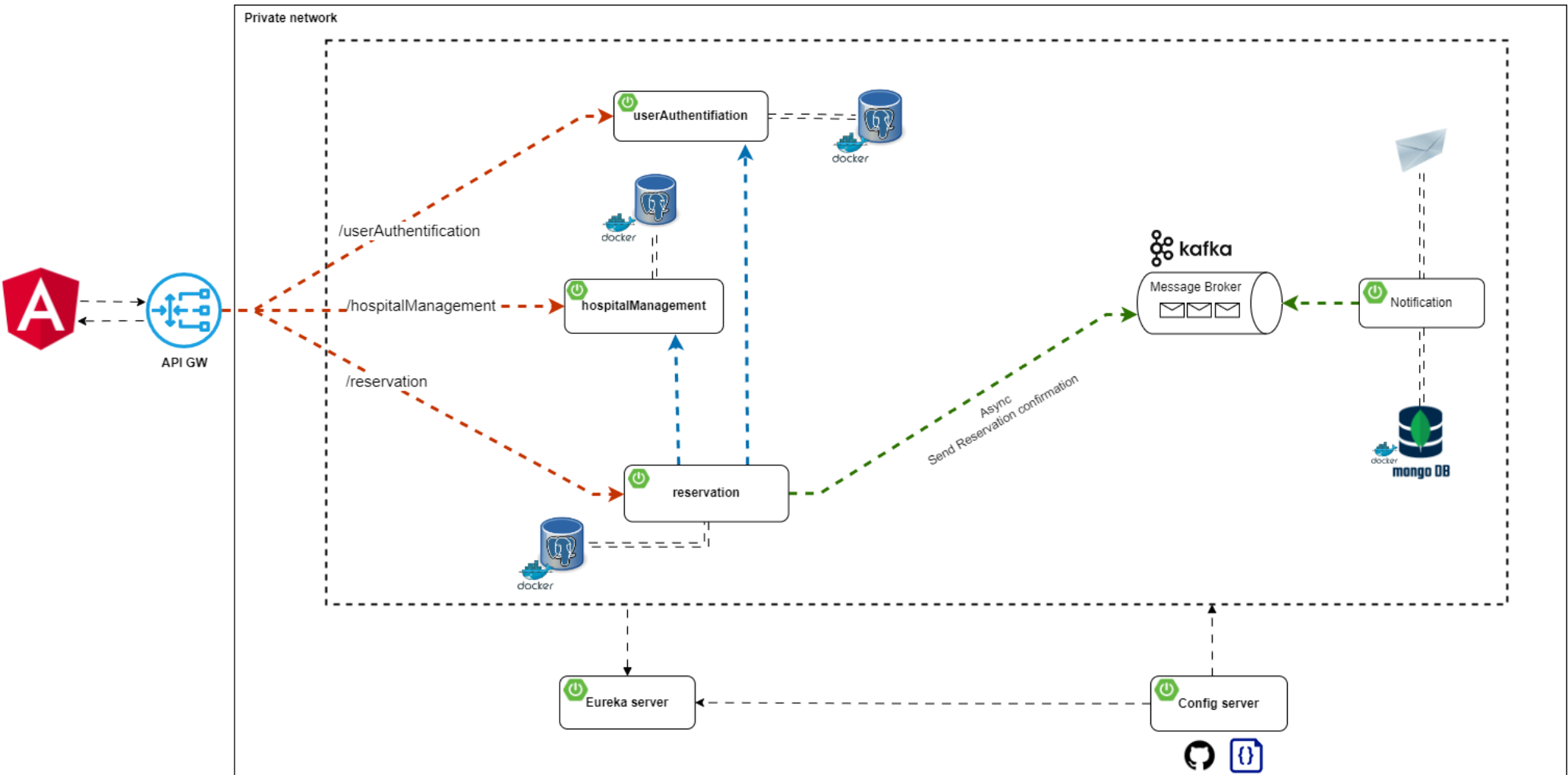
- **Spring Security** : Utilisé pour gérer l'authentification et l'autorisation au niveau des microservices, Spring Security est responsable de la gestion des tokens JWT pour valider les requêtes entrantes.
- **CORS (Cross-Origin Resource Sharing)** : Afin de contrôler les requêtes HTTP provenant de domaines différents de celui du backend, les politiques CORS ont été configurées dans l'API Gateway.
CORS permet de spécifier quels domaines sont autorisés à accéder aux ressources du backend et quels types de requêtes (GET, POST, PUT, etc.) sont permis.
- **Swagger/OpenAPI** : L'utilisation de Swagger pour documenter les endpoints de l'API, associée à l'intégration d'OpenAPI dans le frontend pour générer automatiquement les services HTTP, garantit que ces services respectent parfaitement les spécifications du backend. Cela permet de minimiser les risques de divergence ou de mauvaise implémentation.

2.3. Vision de la Conception de l'Architecture

L'architecture globale du Proof of Concept (PoC) repose sur une approche microservices, choisie pour sa modularité, sa scalabilité, et sa capacité à faciliter l'intégration de nouvelles fonctionnalités. L'idée centrale est de décomposer les fonctions critiques de la plateforme en services indépendants, chacun responsable d'une tâche spécifique. Cette séparation des responsabilités permet d'optimiser la performance, d'améliorer la maintenance, et de faciliter le déploiement continu.

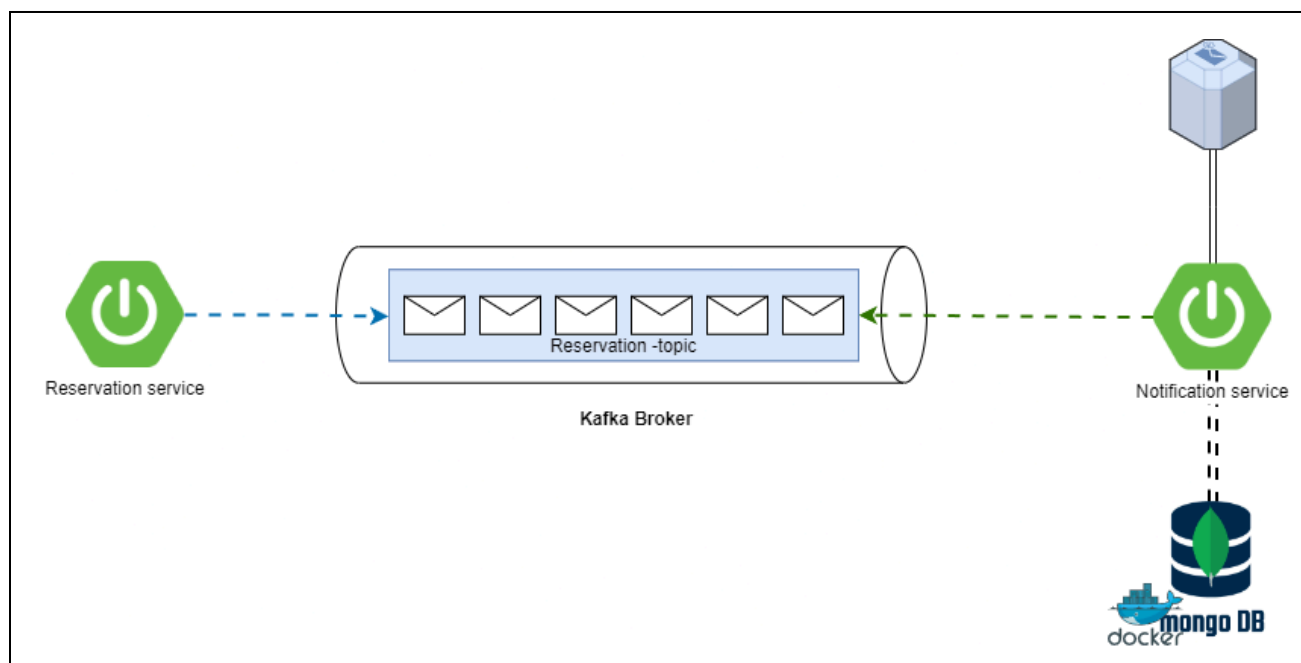
MedHead Consortium

L'architecture de final de la PoC se décompose de la façon suivante :



2.4. Intégration de Kafka

L'utilisation de Kafka dans cette architecture permet d'assurer une communication asynchrone entre les microservices, en particulier pour les processus qui ne nécessitent pas de réponse immédiate, comme l'envoi de notifications. Kafka sert de broker pour transmettre les messages entre le service de réservation et le service de notification, garantissant ainsi la résilience du système face aux pics de charge ou aux interruptions temporaires de certains services.



3. Mise en Œuvre des Microservices

3.1. Présentation Générale

L'architecture microservices offre une modularité et une scalabilité accrues, permettant de développer, déployer et mettre à jour chaque composant indépendamment. Dans cette section, nous présentons les microservices conçus pour ce projet, en détaillant leur rôle, leurs fonctionnalités, et leurs interactions.

3.2. Détails des Microservices

3.2.1. User Authentication Microservice

Rôle :

- Ce microservice est responsable de l'authentification des utilisateurs et de la gestion des tokens JWT. Il assure également l'enregistrement des utilisateurs et l'activation de leur compte via un email de confirmation.

Fonctionnalités :

- **Inscription des utilisateurs** : Permet aux patients de s'enregistrer sur la plateforme.
- **Authentification via JWT** : Après inscription, l'utilisateur peut se connecter. Un token JWT est généré pour chaque utilisateur authentifié, permettant d'accéder aux services protégés.

Le JWT contient des informations sur l'utilisateur, telles que son rôle, et est signé pour garantir son intégrité

- **Activation du compte** : Après l'inscription, un email contenant un code d'activation est envoyé à l'utilisateur. Le compte n'est activé qu'après validation de ce code..

Fonctionnalités développées, mais Non Encore Implémentées Côté Frontend :

- **Obtenir un utilisateur par ID** : Permet de récupérer les informations d'un utilisateur spécifique en fonction de son ID. Cette fonctionnalité est prête côté backend mais pas encore implémentée côté frontend.
- **Obtenir tous les utilisateurs** : Permet de récupérer la liste complète des utilisateurs. Comme pour la précédente, cette API est en place mais n'est pas encore utilisée côté frontend.

Technologies :

- **Spring Boot** pour le développement du service.
- **Spring Security** pour l'authentification et la gestion des tokens JWT.
- **PostgreSQL** pour le stockage des informations des utilisateurs.
- **MailDev** pour l'envoi des mails d'activation des comptes en développement.
- **Flyway** pour la gestion des migrations de base de données et l'insertion de données fictives.

Interactions :

- Utilise Spring Security pour sécuriser les endpoints et valider les tokens JWT lors de chaque requête

3.2.2. Hospital Management Microservice

Rôle :

- Ce service gère les informations relatives aux hôpitaux et à leurs spécialités. Il permet également la recherche des hôpitaux les plus proches en fonction de la spécialité médicale requise par le patient et de son adresse.

Fonctionnalités Actuelles :

- **Recherche d'Hôpitaux** : Le service permet actuellement aux patients de rechercher les hôpitaux les plus proches offrant la spécialité médicale requise et ayant des lits disponibles. Cette fonctionnalité est pleinement implémentée côté frontend et accessible aux utilisateurs.
- **Intégration avec l'API Google (Geocoding API)** : Le microservice utilise l'API Google [Geocoding API](#) pour obtenir des informations géographiques précises, comme les coordonnées GPS, pour calculer la distance entre la localisation du patient et les hôpitaux disponibles. Cette intégration permet de fournir une liste des hôpitaux les plus proches en fonction de la localisation de l'utilisateur.
- **Migration des Données avec Flyway** : Flyway est utilisé pour gérer les migrations de base de données, y compris l'insertion de données fictives pour les hôpitaux et leurs spécialités. Cela permet de préparer la base de données avec des informations nécessaires pour les tests.

Fonctionnalités développées, mais Non Encore Implémentées Côté Frontend :

MedHead Consortium

- **Trouver tous les hôpitaux par spécialité:** Cette fonctionnalité permet de récupérer la liste complète des hôpitaux offrant une spécialité spécifique. Bien que l'API soit prête côté backend, l'implémentation de cette fonctionnalité n'est pas encore réalisée côté frontend.
- **Ajouter une spécialité à un hôpital :** Cette fonctionnalité permet d'ajouter une spécialité à un hôpital existant. Comme pour la précédente, l'API backend est en place, mais l'interface utilisateur côté frontend pour cette action reste à développer.
- **Mise à jour du nombre de lits :** Permet de mettre à jour le nombre de lits disponibles dans un hôpital. Cette fonctionnalité est disponible côté backend, mais non encore implémentée dans le frontend.
- **Créer, Modifier, Supprimer des hôpitaux et des spécialités :** Bien que ces fonctionnalités soient disponibles côté backend, leur implémentation dans le frontend reste à faire.

Technologies :

- **Spring Boot** pour le développement du service.
- **PostgreSQL** pour le stockage des informations des hôpitaux et leurs spécialités.
- **API Google (Geocoding API)** pour l'intégration des services de géolocalisation.
- **Flyway** pour la gestion des migrations de base de données et l'insertion de données fictives.

Interactions :

- Interagit avec le **Reservation Microservice** pour vérifier la disponibilité des lits lors de la réservation.
- Expose des API REST sécurisées via JWT pour les opérations CRUD et la recherche d'hôpitaux.
- Utilise **Geocoding API** pour calculer les distances géographiques et optimiser la recherche d'hôpitaux en fonction de la localisation des patients.

3.2.3. Microservice de Réservation

Rôle :

- Ce service gère les réservations de lits dans les hôpitaux. Il vérifie la disponibilité des lits, et enregistre les réservations effectuées par les patients pour un hôpital spécifique.

Fonctionnalités Actuelles :

MedHead Consortium

- **Effectuer une réservation** : Cette fonctionnalité permet aux patients de réserver un lit dans un hôpital sélectionné.
- **Obtenir toutes les réservations** : Permet de récupérer la liste complète des réservations effectuées.

Technologies :

- **Spring Boot** pour le développement du service.
- **Kafka** pour publier les événements de confirmation de réservation.
- **PostgreSQL** pour le stockage des informations de réservation.

Interactions :

- Publie des messages sur Kafka pour notifier le **Notification Microservice** de la confirmation de réservation.
- Utilise Spring Security pour sécuriser les endpoints et valider les tokens JWT lors de chaque requête.
- Lorsqu'une réservation est effectuée, le microservice de réservation interagit avec le microservice d'authentification pour récupérer les informations sur le patient qui a initié la réservation. Cela garantit que la réservation est associée à l'utilisateur connecté.
- Le microservice de réservation interroge le microservice de gestion des hôpitaux pour modifier le nombre de lits disponibles à la fin de la réservation.

3.2.4. Notification Microservice

Rôle :

- Ce service est chargé d'envoyer des notifications par email, principalement pour confirmer les réservations de lits.

Fonctionnalités :

- **Envoi de notifications** : Écoute les événements sur le topic Kafka et envoie des emails de confirmation de réservation aux patients.
- **Gestion des modèles d'email** : Les modèles d'emails sont utilisés pour personnaliser les messages envoyés.

Technologies :

- **Spring Boot** pour le développement du service.
- **Kafka** pour la gestion des événements asynchrones.

MedHead Consortium

- **MongoDB** pour stocker les logs et les configurations des notifications.
- **MailDev** pour l'envoi des mails de confirmation des réservations en développement.

Interactions :

- Consomme les messages du **Reservation Microservice** via Kafka pour déclencher l'envoi d'emails.

3.2.5. API Gateway

Rôle :

- L'API Gateway sert de point d'entrée unique pour toutes les requêtes du frontend. Elle centralise le routage vers les microservices backend, gère l'authentification, et applique les politiques de sécurité.

Fonctionnalités :

- **Routage des requêtes** : Transmet les requêtes du frontend aux microservices appropriés.
- **Gestion de la sécurité** : Vérifie les tokens JWT pour valider les accès aux endpoints sécurisés.
- **Contrôle CORS** : Applique les règles CORS pour limiter l'accès aux ressources backend depuis des domaines non autorisés.

Technologies :

- **Spring Cloud Gateway** pour la gestion du routage et de la sécurité.
- **Spring Security** pour la gestion des tokens JWT et des politiques CORS.

Interactions :

- Interagit avec tous les microservices pour assurer le routage correct et appliquer les politiques de sécurité.

3.2.6. Discovery Server (Eureka)

Rôle :

- Le Discovery Server, basé sur Eureka, permet à chaque microservice de s'enregistrer dynamiquement et de découvrir les autres services sans configuration statique. Cela facilite l'élasticité et l'évolutivité de l'application, car les services peuvent être ajoutés ou retirés dynamiquement.

Fonctionnalités :

- **Service Registry** : Tous les microservices s'enregistrent auprès du Discovery Server pour être découverts par les autres services.
- **Load Balancing** : Permet de répartir la charge entre plusieurs instances d'un même service.

Technologies :

- **Netflix Eureka** pour la gestion de la découverte de services.

Interactions :

- Tous les microservices s'enregistrent auprès du Discovery Server pour permettre une découverte et un routage dynamiques.

3.2.7. Config Server

Rôle :

- Le Config Server centralise la gestion des configurations pour tous les microservices. Il permet de gérer les configurations de manière externe et de les appliquer dynamiquement sans nécessiter de redéploiement des services.

Fonctionnalités :

- **Centralisation des configurations** : Les configurations des microservices sont externalisées et centralisées.
- **Mise à jour dynamique** : Les configurations peuvent être mises à jour dynamiquement sans redéploiement des microservices.

Technologies :

- **Spring Cloud Config** pour la gestion centralisée des configurations.

Interactions :

- Tous les microservices récupèrent leurs configurations auprès du Config Server au démarrage et peuvent être mis à jour dynamiquement en cas de modification des configurations.

3.3. Orchestration et Déploiement avec Docker Compose

Dans cette phase du Proof of Concept (PoC), l'accent a été mis sur l'implémentation et l'orchestration des services essentiels pour la plateforme.

Docker Compose a été mis en place dès cette phase du PoC. Docker Compose permet de démarrer et de configurer l'ensemble des services essentiels avec une seule commande, assurant ainsi une cohérence et une simplicité dans la gestion de l'environnement.

Configuration Docker Compose

Le fichier `docker-compose.yml` configure les services suivants :

- **PostgreSQL et pgAdmin** : Pour la gestion des données relationnelles.
- **Kafka et Zookeeper** : Pour la gestion des messages asynchrones.
- **MongoDB et Mongo-Express** : Pour la gestion des données non structurées.
- **MailDev** : Pour la gestion des notifications par email en développement.

Chaque service est configuré avec ses ports, ses volumes pour la persistance des données, et ses dépendances, garantissant ainsi un déploiement fluide et intégré de l'ensemble de l'infrastructure.























Création et Initialisation des Bases de Données :

Lors du démarrage du service PostgreSQL via Docker Compose, les bases de données nécessaires pour les microservices sont créées et initialisées. Cette configuration permet de préparer l'environnement de manière cohérente et automatisée assurant ainsi que les microservices peuvent démarrer avec une base de données prête à l'emploi.

Réseau et Volumes

Docker Compose configure un réseau dédié (`microservices-net`) pour permettre une communication fluide entre les services. Des volumes persistants sont également utilisés pour garantir que les données critiques ne sont pas perdues lors des redémarrages des conteneurs.

Ci-dessous le tableau de bord Docker desktop des conteneurs actuellement en cours d'exécution pour notre projet "**medheadapp_poc**".

Name	Image	Status	Port(s)	CPU (%)	Last started
 medheadapp_poc		Running (7/7)		337.29%	33 seconds ago
 pgadmin 1bd7a2688743 	dpage/pgadmin4	Running	5050:80 	96.71%	37 seconds ago
 mongodb b45852991bf4 	mongo	Running	27017:27017 	2.56%	38 seconds ago
 zookeeper 5fa98dcdbdf39 	confluentinc/cp-zookeeper:latest	Running	22181:2181 	0.6%	37 seconds ago
 postgres 1f43ae9a41b4 	postgres	Running	5432:5432 	0.04%	37 seconds ago
 mail-dev 08e3f94f5347 	maildev/maildev	Running	1025:1025  Show all ports (2)	7.72%	37 seconds ago
 mongo-express 942146caa458 	mongo-express	Running	8081:8081 	0.01%	33 seconds ago
 kafka 01bff24ad4ec 	confluentinc/cp-kafka:latest	Running	9092:9092 	229.65%	33 seconds ago

4. Tests et Validation

4.1. Introduction aux Stratégies de Test

Les tests sont un élément clé pour garantir la qualité et la robustesse du système, surtout dans le contexte d'une architecture microservices. Dans ce Proof of Concept (PoC), une attention particulière a été portée aux tests du microservice d'authentification des utilisateurs. En raison du caractère exploratoire du PoC, tous les tests nécessaires n'ont pas encore été implémentés pour les autres microservices, mais ils sont prévus dans les étapes ultérieures du projet.

Les types de tests effectués incluent :

- **Tests Unitaires et d'Intégration pour le Microservice d'Authentification :** Validation des fonctionnalités critiques du microservice d'authentification, avec des tests couvrant le contrôleur, le service et le mapper.
- **Tests API avec Postman :** Tests manuels des endpoints pour tous les microservices afin de vérifier leur bon fonctionnement.
- **Tests de Montée en Charge :** Validation de la performance pour la recherche d'hôpitaux.

MedHead Consortium

- **Tests End-to-End (E2E)** : Réalisés côté frontend pour valider les scénarios utilisateurs de bout en bout.

4.2. Tests Unitaires et d'Intégration

Les tests unitaires et d'intégration ont été principalement réalisés pour le microservice d'authentification des utilisateurs, dans le cadre du respect des principes BDD. Ces tests ont été essentiels pour valider le bon fonctionnement de chaque composant du microservice, de manière isolée (tests unitaires) et en interaction avec d'autres composants (tests d'intégration).

Microservice d'Authentification des Utilisateurs :

- **Contrôleur** : Des tests ont été réalisés pour valider les actions du contrôleur, telles que l'inscription, la connexion, l'activation du compte, la récupération des utilisateurs, et la suppression d'utilisateurs. Par exemple, le test `testRegister()` vérifie que l'inscription d'un utilisateur se déroule comme prévu.
- **Service** : Le service d'authentification a été testé pour s'assurer que les logiques métiers essentielles, comme la génération et la validation des JWT, l'activation de compte, et la gestion des utilisateurs, fonctionnent correctement. Le test `shouldAuthenticateUser()` valide, par exemple, que l'authentification d'un utilisateur retourne un token JWT valide.
- **Mapper** : Les mappers, responsables de la transformation des données entre les différentes couches, ont été testés pour garantir la cohérence des données lors des opérations de mapping.
Par exemple, le test `shouldMapRegistrationRequestToUser()` vérifie que les données d'une requête d'inscription sont correctement transformées en objet `User`.

Résultats des tests unitaires :

```
3568 [INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.181 s -- in com.example.medhead.controller.AuthControllerTest
3569 [INFO] Running com.example.medhead.services.AuthServiceTest
3570 [INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.836 s -- in com.example.medhead.services.AuthServiceTest
3571 [INFO] Running com.example.medhead.mapper.UserMapperTest
3572 [INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.049 s -- in com.example.medhead.mapper.UserMapperTest
3573 [INFO]
3574 [INFO] Results:
3575 [INFO]
3576 [INFO] Tests run: 19, Failures: 0, Errors: 0, Skipped: 0
3577 [INFO]
3578 [INFO] -----
3579 [INFO] BUILD SUCCESS
3580 [INFO] -----
3581 [INFO] Total time: 25.044 s
3582 [INFO] Finished at: 2024-08-13T23:33:49Z
3583 [INFO] -----
```

Utilisation de H2 :

Le choix de H2 en tant que base de données en mémoire pour les tests offre plusieurs avantages, notamment la rapidité d'exécution des tests et la simplicité de configuration. H2 permet également de simuler une base de données réelle sans avoir besoin d'installer et de configurer un système de base de données complet, ce qui est particulièrement utile pour les tests unitaires.

Outils Utilisés :

- **JUnit** : Utilisé pour l'exécution des tests unitaires.
- **Mockito** : Employé pour simuler les dépendances et les interactions complexes.

Note Importante : Bien que les tests unitaires et d'intégration soient bien en place pour le microservice d'authentification, les autres microservices nécessitent également des tests similaires pour assurer leur robustesse. Ces tests seront une priorité dans les phases suivantes du projet pour garantir que chaque microservice fonctionne correctement, tant de manière isolée qu'en interaction avec les autres services.

4.3. Tests API avec Postman

En complément des tests unitaires, tous les endpoints des microservices ont été testés manuellement avec Postman pour vérifier leur bon fonctionnement. Ces tests garantissent que les API REST exposées par chaque microservice retournent les résultats attendus et gèrent correctement les différentes conditions.

Scénarios Testés :

MedHead Consortium

- **User Authentication Microservice** : Tests des opérations de création d'utilisateur, de connexion, et de gestion des utilisateurs.

Requête d'authentification API réussie avec retour de jeton et informations utilisateur :

The screenshot displays a REST client interface for a POST request to the endpoint `{{api_url}}:{{api_port}}/api/auth/authenticate`. The request body is a JSON object containing email and password. The response is a 200 OK status with a JSON body containing a token and user details.

Request:

```
1 {
2   "email": "Emili@gmail.com",
3   "password": "123456789"
4 }
```

Response:

```
1 {
2   "token": "eyJhbGciOiJIUzI1NiJ9.eyJmdWxsTmFtZSI6IktawxpIHZpdmVzIiwic3ViIjojRw1pbGIAZ21hawWuY29tIiwiaWF0IjoxNzIzODI2NzUwLCJleHAiOjE3MjQ0MzE1NTAsImF1dGhvcm10awVzIjpbI1BhdGllbnQiXX0.InHYupcps1hQziqczEpcwcFTwOK4g1rj7VPmJue18_c",
3   "user": {
4     "id": 3,
5     "nom": "vives",
6     "prenom": "Emili",
7     "dateNaissance": "1989-01-22",
8     "email": "Emili@gmail.com",
9     "sexe": "Homme",
10    "adresse": "Espagne Barcelone",
11    "numero": "3369856321",
12    "roles": [
13      "Patient"
14    ],
15    "accountLocked": false,
16    "enabled": true
17  }
18 }
```

MedHead Consortium

- **Hospital Management Microservice** : Validation des opérations de recherche d'hôpitaux, d'ajout de spécialités, et de mise à jour des lits.

Requête API pour trouver les plus proches hôpitaux avec des lits disponibles dans une spécialité spécifique :

The screenshot shows a REST client interface with the following details:

- Request Method:** GET
- URL:** `http://localhost:8222/api/hospital/nearest?address=34 Quai d'Austerlitz&specialty=Cardiologie`
- Query Params:**

Key	Value	Description
address	34 Quai d'Austerlitz	
specialty	Cardiologie	
- Status:** 200 OK
- Time:** 1093 ms
- Size:** 1.61 KB

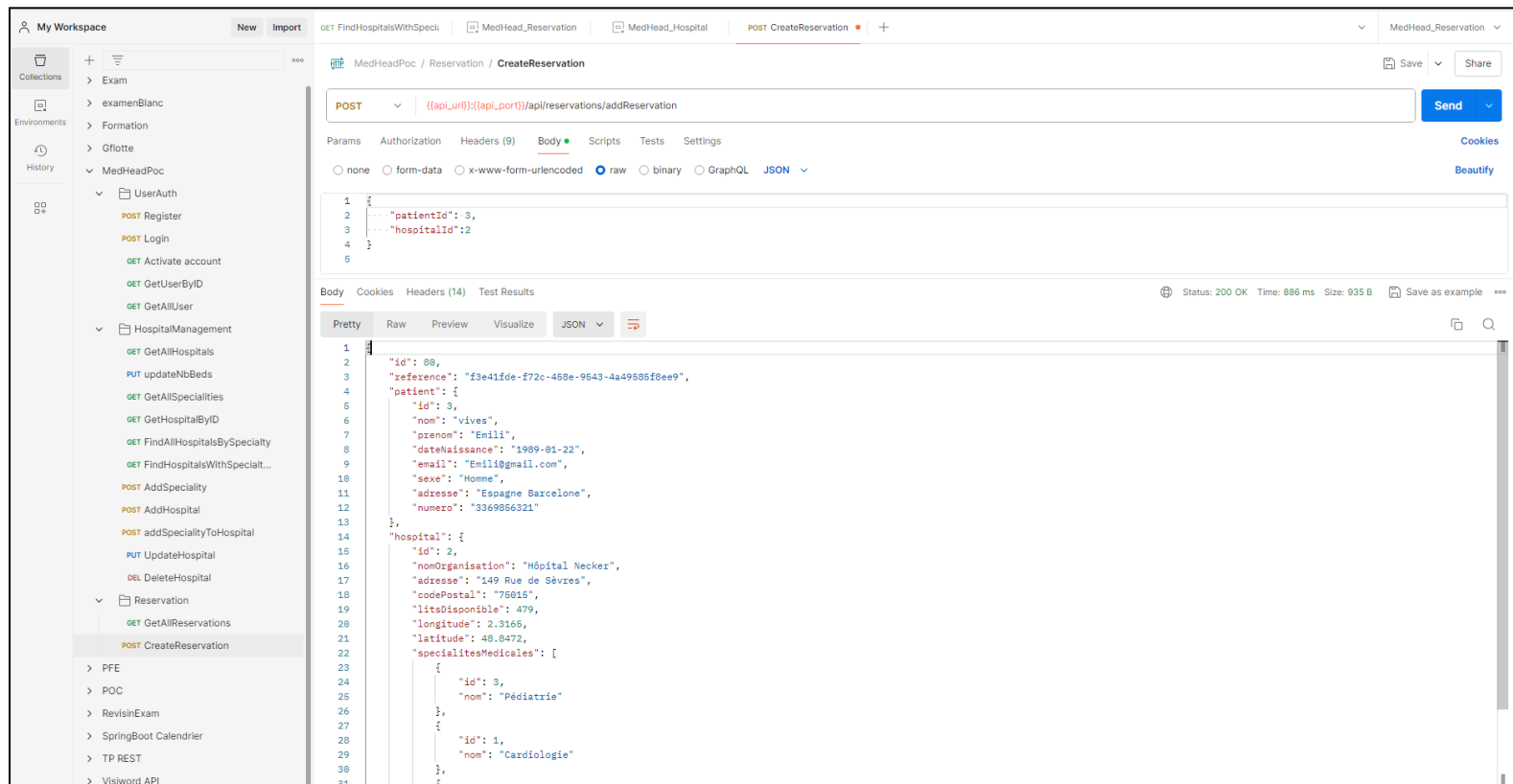
The response body is a JSON array containing one object:

```
[
  {
    "id": 3,
    "nomOrganisation": "Hôpital Pitié-Salpêtrière",
    "adresse": "47-83 Boulevard de l'Hôpital",
    "codePostal": "75013",
    "specialitesMedicales": [
      {
        "id": 1,
        "nom": "Cardiologie"
      },
      {
        "id": 10,
        "nom": "Psychiatrie"
      },
      {
        "id": 2,
        "nom": "Neurologie"
      }
    ],
    "litsDisponible": 977,
    "longitude": 2.3644,
    "latitude": 48.8399,
    "distance": 0.4210263310396185
  }
]
```

MedHead Consortium

- **Reservation Microservice** : Vérification des opérations de réservation et de gestion des réservations.

Création d'une réservation d'hôpital via une requête API:



Résultats :

- Tous les endpoints testés avec Postman ont fonctionné comme prévu, validant ainsi leur bon fonctionnement dans les scénarios couverts.

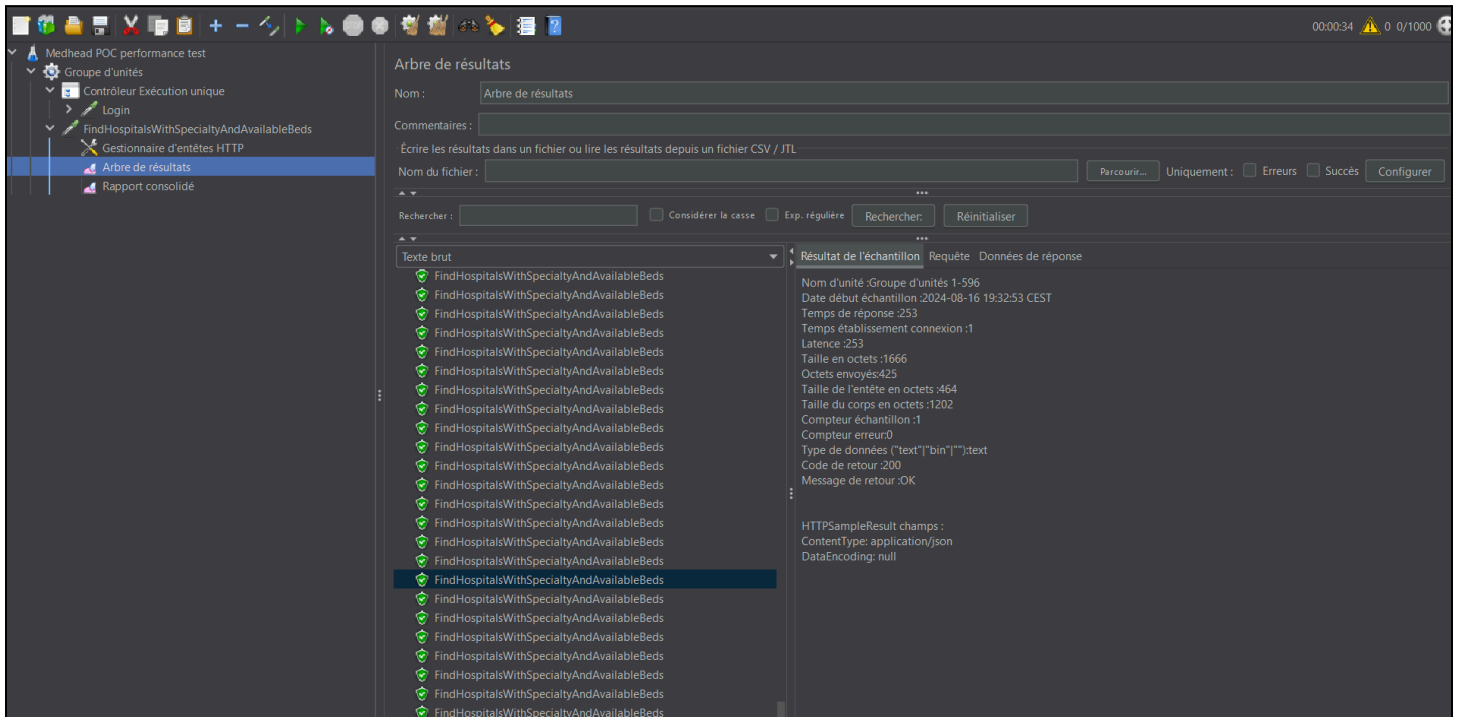
4.4. Tests de Montée en Charge et de performance

Un test de montée en charge a été réalisé pour évaluer la performance du microservice de gestion des hôpitaux, en particulier pour la fonctionnalité de recherche.

Scénario Testé :

- **Recherche d'Hôpitaux** : 1000 requêtes par seconde ont été envoyées pour évaluer la capacité du service à répondre sous une charge élevée.

Résultats de tests de montée en charge JMeter pour l'API FindHospitalsWithSpecialtyAndAvailableBeds :



Résultats :

- Les résultats ont démontré que notre système est parfaitement capable de gérer une telle charge sans compromettre la performance ou la stabilité, attestant ainsi de la robustesse et de l'efficacité de notre solution dans des conditions de trafic élevé.

Résultats des Tests de Performance

medhead_poc_charge_test.jmx (C:\Users\zanto\OneDrive\Bureau\Master_Expert_En_Inj_Logiciel\Block4\MedHeadApp_PoC\jmeter\medhead_poc_charge_test.jmx) - Apache JMeter (5.6.3)

00:00:35 0 0/1000

Rapport consolidé

Nom : Rapport consolidé

Commentaires :

Écrire les résultats dans un fichier ou lire les résultats depuis un fichier CSV / JTL

Nom du fichier : Parcourir... Uniquement : ☐ Erreurs ☐ Succès Configurer

Libellé	# Echantillons	Moyenne	Min	Max	Ecart type	% Erreur ↓	Débit	Ko/sec reçus	Ko/sec émis	Moy. octets
FindHospitals...	1000	610	126	2522	480.60	0,00%	28.8/sec	46,61	12,13	1660,0
TOTAL	1000	610	126	2522	480.60	0,00%	28.8/sec	46,61	12,13	1660,0

Résultats :

Le test de performance réalisé avec JMeter sur la fonctionnalité de recherche d'hôpitaux a permis de simuler une charge de 1000 utilisateurs simultanés. Les résultats montrent que le système a pu gérer cette charge de manière efficace, avec les métriques suivantes :

- **Nombre total d'échantillons** : 1000 requêtes
- **Temps de réponse moyen** : 610 ms
- **Temps de réponse minimal** : 126 ms
- **Temps de réponse maximal** : 2522 ms
- **Taux d'erreur** : 0%
- **Débit** : 28.8 requêtes par seconde

Ces résultats sont encourageants et montrent que l'architecture du système est capable de gérer une charge significative tout en maintenant des temps de réponse raisonnables. Cependant, il est recommandé de poursuivre les optimisations pour réduire le temps de réponse, en particulier sous des charges extrêmes.

4.5. Tests End-to-End (E2E)

Les tests E2E ont été réalisés pour valider les flux utilisateurs complets, de l'interface frontend aux microservices backend, assurant ainsi que l'ensemble du système fonctionne correctement du point de vue de l'utilisateur final.

Outils Utilisés :

- **Cypress** pour l'automatisation des tests E2E.

Scénarios Testés :

- **Connexion (admin/patient)** : Vérification que les utilisateurs peuvent se connecter en fonction de leur rôle.
- **Recherche et Réservation** : Validation de la recherche d'hôpitaux et de la réservation de lits.
- **Consultation des Réservations** : Test de la capacité de l'administrateur à consulter la liste des réservations effectuées.
- **Consultation des hôpitaux** : Vérifie que l'administrateur peut voir la liste de tous les hôpitaux.
- **Accès interdits** : Vérifie que l'accès à certaines pages est interdit aux utilisateurs non autorisés.
- **Echec de la connexion** : Vérifie les scénarios d'échec de connexion avec des identifiants incorrects.

Résultats des tests automatisés côté front-end avec Cypress

Spec	Tests	Passing	Failing	Pending	Skipped
✓ ListReservationAdmin.cy.ts	00:04	2	2	-	-
✓ forbiddenAccess.cy.ts	00:02	1	1	-	-
✓ listAllHospitalsForAdmin.cy.ts	00:04	1	1	-	-
✓ login-admin.cy.ts	00:04	2	2	-	-
✓ login-failed.cy.ts	00:04	4	4	-	-
✓ login-patient.cy.ts	00:03	2	2	-	-
✓ reservation.cy.ts	00:03	1	1	-	-
✓ searchNearstHospital.cy.ts	00:05	3	3	-	-
✓ All specs passed!	00:32	16	16	-	-

Conclusion :

Les résultats des tests montrent que le système développé dans ce PoC répond aux attentes en termes de fonctionnalité, de performance, et de sécurité.

Les tests unitaires, les tests API avec Postman, les tests de montée en charge, et les tests E2E ont permis de valider les fonctionnalités critiques mises en œuvre.

Toutefois, pour garantir une robustesse optimale dans un environnement de production, il est impératif de compléter les tests unitaires et d'intégration pour tous les microservices. Cette prochaine étape est cruciale pour le succès à long terme du projet.

5. Pipelines CI/CD avec GitHub Actions

5.1. Introduction

Dans le cadre de ce Proof of Concept (PoC), deux pipelines d'intégration continue (CI) ont été mis en place en utilisant GitHub Actions : l'un pour le frontend développé en Angular et l'autre pour le backend constitué de microservices développés en Spring Boot.

L'objectif de ces pipelines est de garantir que chaque modification apportée au code source est automatiquement testée et compilée, afin de détecter les erreurs le plus tôt possible et de maintenir une qualité de code élevée tout au long du cycle de développement.

Ces pipelines sont déclenchés à chaque **push** sur n'importe quelle branche du dépôt, assurant ainsi une intégration continue de toutes les modifications apportées au projet. Un des avantages clés de ces pipelines est qu'ils permettent d'automatiser les tests, ce qui assure une vérification rapide et systématique de chaque modification.

5.2. Pipeline CI pour le Backend

Le pipeline CI pour le backend est conçu pour compiler et tester les différents microservices qui composent l'architecture du projet. Ce pipeline assure que chaque microservice fonctionne correctement et peut être déployé sans problème.

Étapes du Pipeline :

1. **Compilation :**

- Chaque microservice (Config Server, Discovery Service, User Authentication, Hospital Management, Reservation, Notification, Gateway API) est compilé en utilisant Maven.
- Cette étape garantit que le code de chaque microservice peut être compilé sans erreur.

2. **Exécution des Tests Unitaires :**

- Après la compilation, des tests unitaires sont automatiquement exécutés pour chaque microservice. Ces tests sont essentiels pour vérifier le bon fonctionnement des composants internes de chaque microservice.

3. **Construction des Microservices :**

- Une fois les tests réussis, chaque microservice est empaqueté. Cette étape génère des fichiers JAR qui peuvent ensuite être déployés sur un environnement de production ou de test.

5.3. Pipeline CI pour le Frontend

Le pipeline CI pour le frontend est conçu pour compiler et tester l'application Angular, s'assurant que l'interface utilisateur fonctionne comme prévu et que les modifications du code n'introduisent pas de régressions.

Étapes du Pipeline :

1. Installation des Dépendances :

- Le code source est d'abord extrait du dépôt GitHub, puis les dépendances du projet sont installées pour garantir que le projet utilise une version cohérente des packages.

2. Exécution des Tests Unitaires :

- Les tests unitaires du frontend sont automatiquement exécutés. Cette automatisation permet de vérifier que les composants Angular fonctionnent correctement sans intervention manuelle, assurant une vérification rapide des modifications.

3. Compilation du Frontend :

- Si les tests sont réussis, l'application Angular est compilée. Cette étape génère les fichiers de production prêts à être déployés.

4. Tests End-to-End (E2E) :

- Une fois l'application compilée, le serveur Angular est démarré et les tests E2E sont exécutés à l'aide de Cypress.
- Les tests E2E simulent des interactions utilisateur avec l'application, validant que les fonctionnalités fonctionnent de bout en bout.

5.4. Avantages de l'Implémentation des Pipelines CI

Automatisation des Tests et de la Validation :

- Les pipelines CI/CD automatisent non seulement le processus de test, mais aussi la compilation, garantissant que le code reste stable et prêt à être déployé à chaque étape du développement.

Prêt pour la Production :

- Ces pipelines sont configurés pour assurer que chaque version du code validée est prête pour le déploiement en production, facilitant ainsi une transition rapide et sûre des développements vers l'environnement de production.

Détection Précoce des Erreurs :

- Les erreurs sont détectées rapidement, ce qui permet de les corriger avant qu'elles ne se propagent dans le système, assurant ainsi la qualité continue du code.

Amélioration de l'Efficacité :

- En automatisant les tâches répétitives, les pipelines CI/CD permettent à l'équipe de développement de se concentrer sur des tâches à plus forte valeur ajoutée, tout en assurant un cycle de développement plus rapide et plus fiable.

6. Illustrations de l'Interface Utilisateur et de la Documentation API

6.1. Accès en tant qu'Administrateur et Patient

Dans ce Proof of Concept (PoC), l'accès à la plateforme peut se faire sous deux rôles différents : Administrateur et Patient. Ces comptes ont été créés et insérés dans la base de données en utilisant Flyway, pour faciliter les tests.

Accès en tant qu'Administrateur

Pour tester les fonctionnalités dédiées à l'administrateur, un compte administrateur a été préconfiguré via un script de migration Flyway. Voici les informations de connexion pour l'administrateur :

- **Email** : admin@example.com
- **Mot de passe** : admin_password

En tant qu'administrateur, vous pouvez accéder aux fonctionnalités suivantes :

- Consultation de la liste des réservations effectuées par les patients.
- Gestion des hôpitaux et de leurs spécialités.

Accès en tant que Patient

De la même manière, un compte patient a été créé pour tester les fonctionnalités réservées aux utilisateurs patients. Ce compte est également inséré via un script de migration Flyway. Voici les informations de connexion pour le patient :

- **Email** : patient@example.com
- **Mot de passe** : patient_password

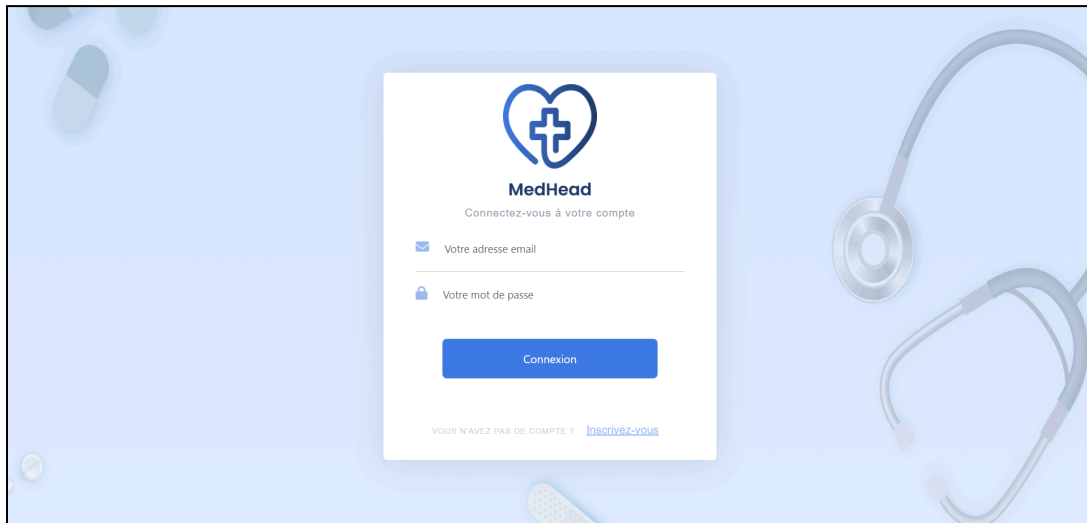
En tant que patient, vous pouvez :

- Rechercher des hôpitaux par spécialité et localisation.
- Réserver un lit dans un hôpital sélectionné.
- Recevoir un email de confirmation pour chaque réservation effectuée.

6.2. Interface Utilisateur Angular

Page de Connexion :

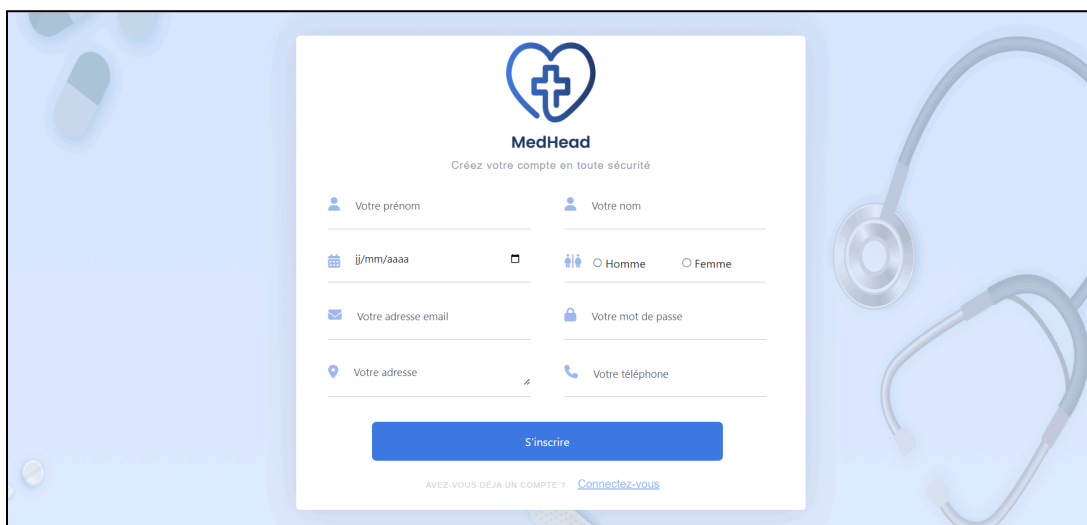
- Cette capture montre la page de connexion où les utilisateurs peuvent entrer leurs identifiants pour accéder à la plateforme



The screenshot shows the MedHead login interface. At the top is the MedHead logo, a heart with a cross inside. Below the logo is the text "Connectez-vous à votre compte". There are two input fields: "Votre adresse email" and "Votre mot de passe". Below these fields is a blue button labeled "Connexion". At the bottom, there is a link that says "VOUS N'AVEZ PAS DE COMPTE ? [Inscrivez-vous](#)". The background is light blue with medical icons like pills and a stethoscope.

Page d'Inscription :

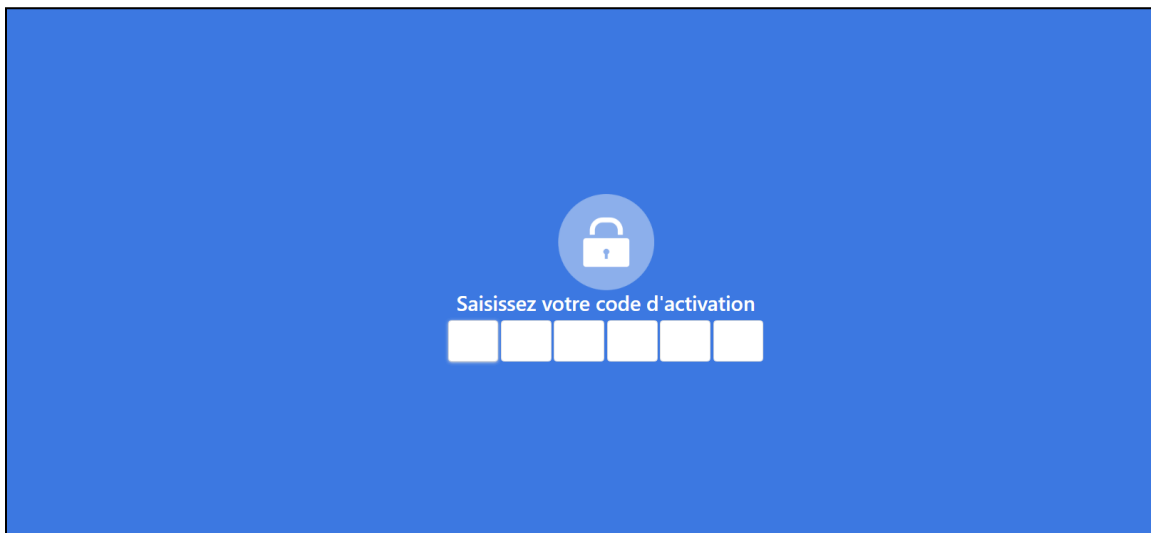
- Cette capture illustre la page d'inscription, où les nouveaux patients peuvent créer un compte en fournissant leurs informations personnelles.



The screenshot shows the MedHead registration interface. At the top is the MedHead logo, a heart with a cross inside. Below the logo is the text "Créez votre compte en toute sécurité". There are several input fields: "Votre prénom", "Votre nom", "jj/mm/aaaa" (with a calendar icon), "Votre adresse email", "Votre adresse", "Votre mot de passe", and "Votre téléphone". There are also radio buttons for "Homme" and "Femme". Below these fields is a blue button labeled "S'inscrire". At the bottom, there is a link that says "AVEZ-VOUS DÉJÀ UN COMPTE ? [Connectez-vous](#)". The background is light blue with medical icons like pills and a stethoscope.

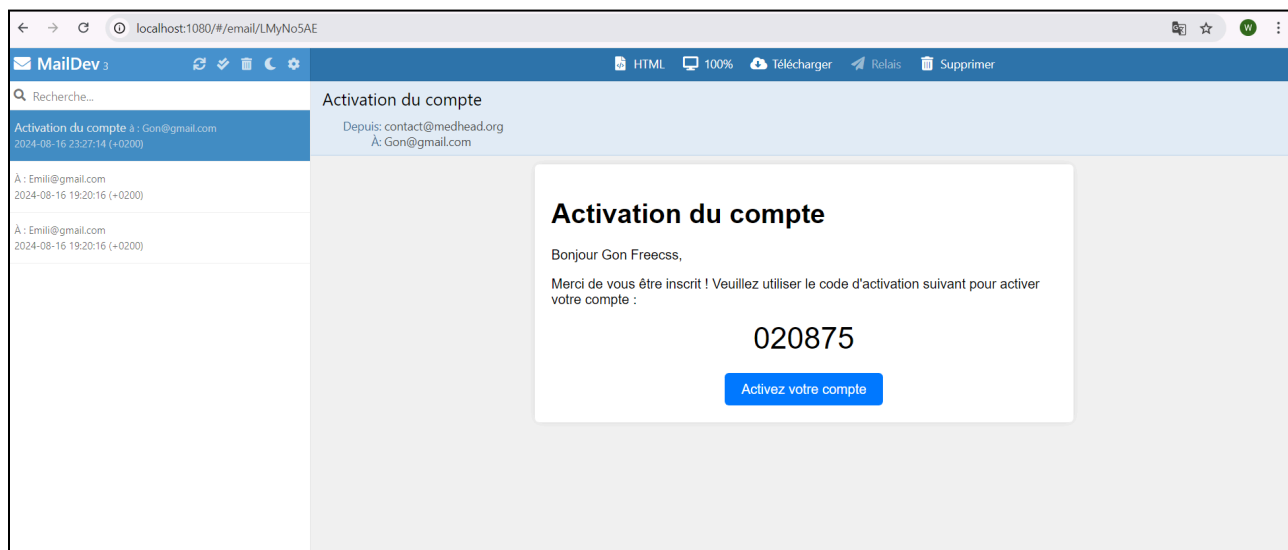
Page d'Activation du Compte :

- Après l'inscription, le patient reçoit un email contenant un code d'activation. Il doit entrer ce code sur la page d'activation pour activer son compte.



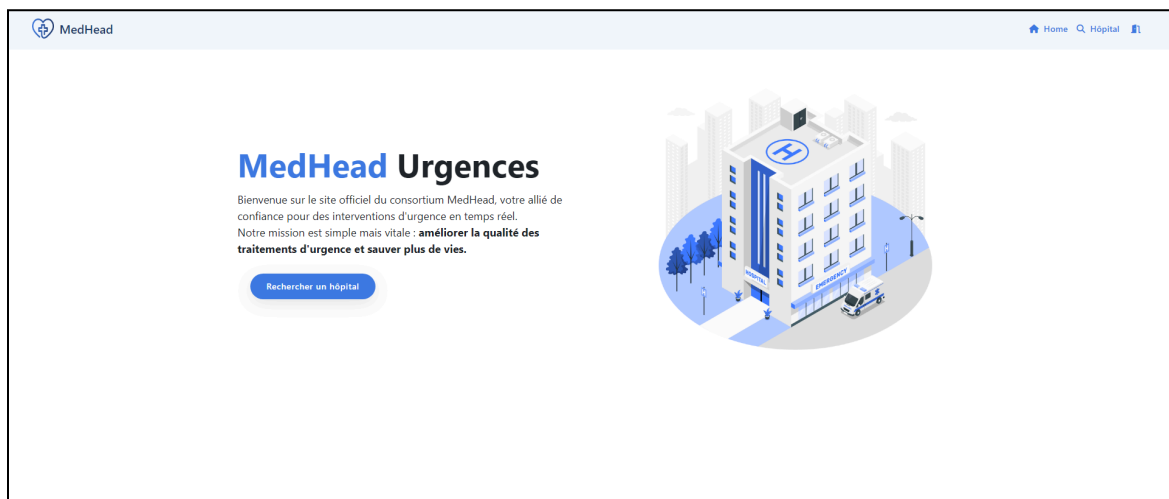
Accès à l'Interface Maildev :

- Étant donné que le projet est en phase de développement, l'email contenant le code d'activation est envoyé à l'outil Maildev (Il s'agit d'un serveur SMTP (Simple Mail Transfer Protocol) simulé). Pour récupérer ce code, il suffit d'accéder à l'interface Maildev via l'URL suivante : <http://localhost:1080/#/>.



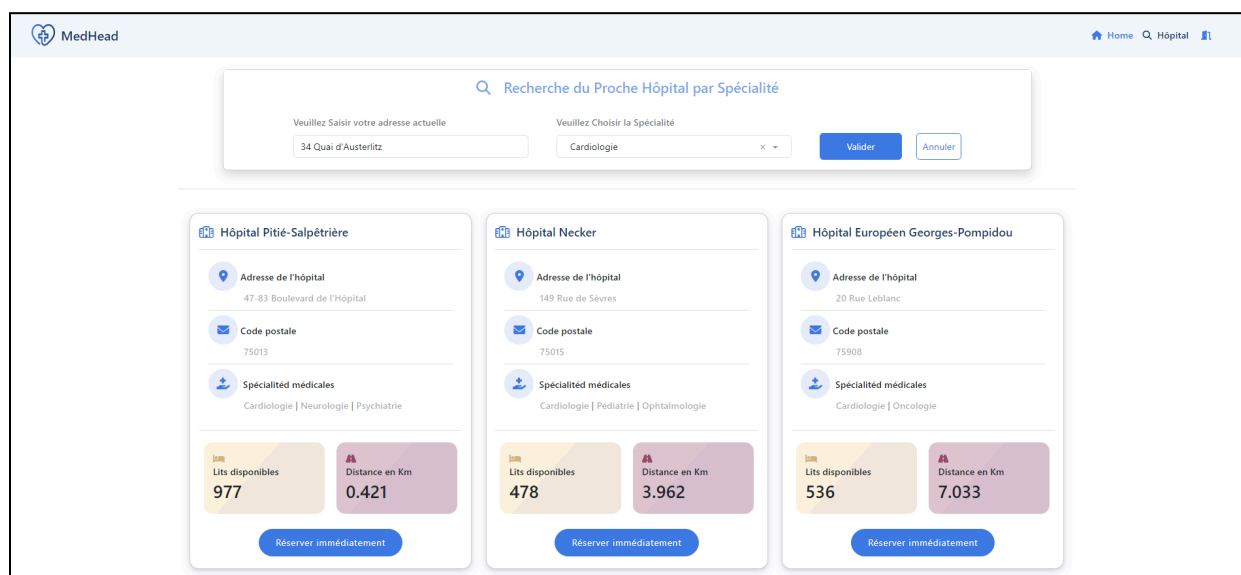
Page d'Accueil pour le Patient :

- Une fois connecté et le compte activé, le patient est dirigé vers la page d'accueil. Cette page offre un accès rapide aux principales fonctionnalités, telles que la recherche d'hôpitaux.



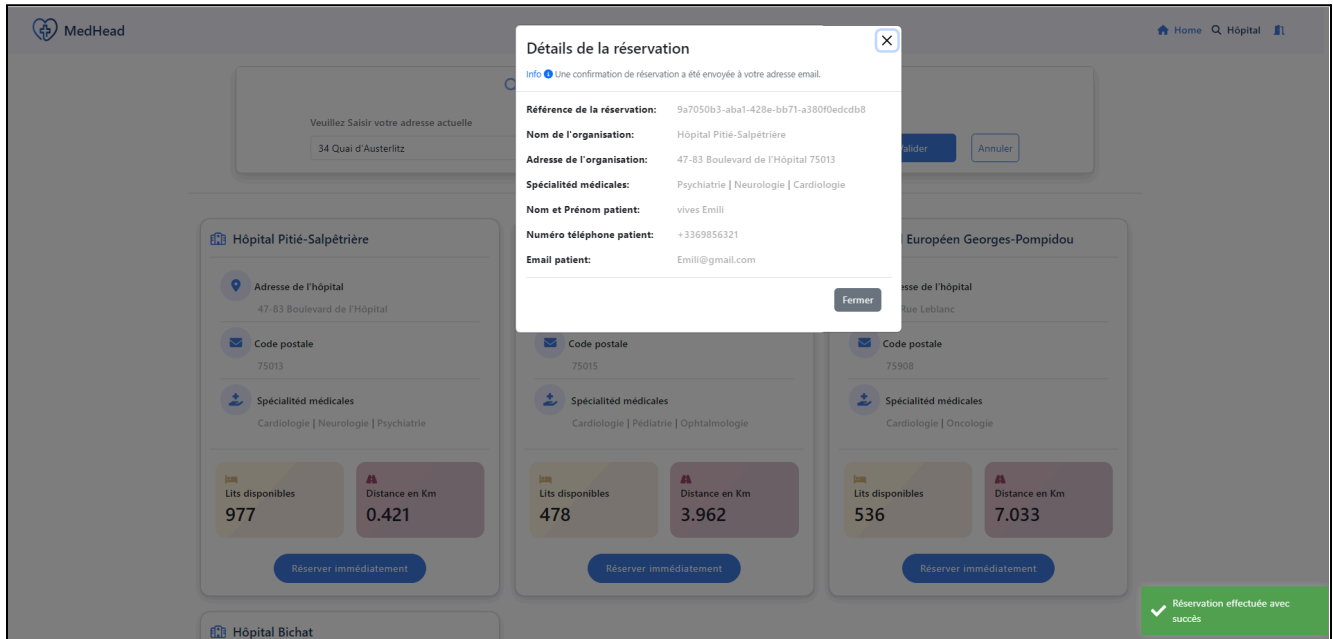
Page de Recherche d'Hôpitaux :

- Sur cette page, le patient peut rechercher les hôpitaux selon une spécialité donnée et son emplacement. La capture d'écran montre les résultats d'une recherche effectuée, affichant les hôpitaux les plus proches correspondant aux critères spécifiés.

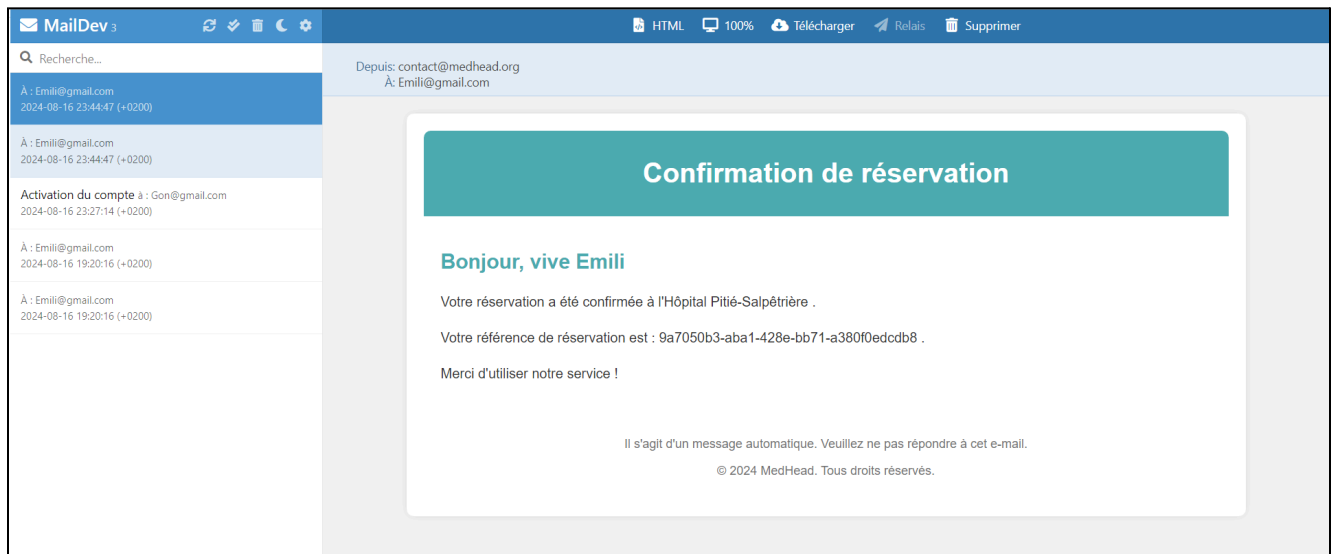


Modal de Confirmation de Réservation :

- Après avoir cliqué sur le bouton de réservation, un modal s'affiche avec les détails de la réservation, et un email de confirmation est automatiquement envoyé au patient.

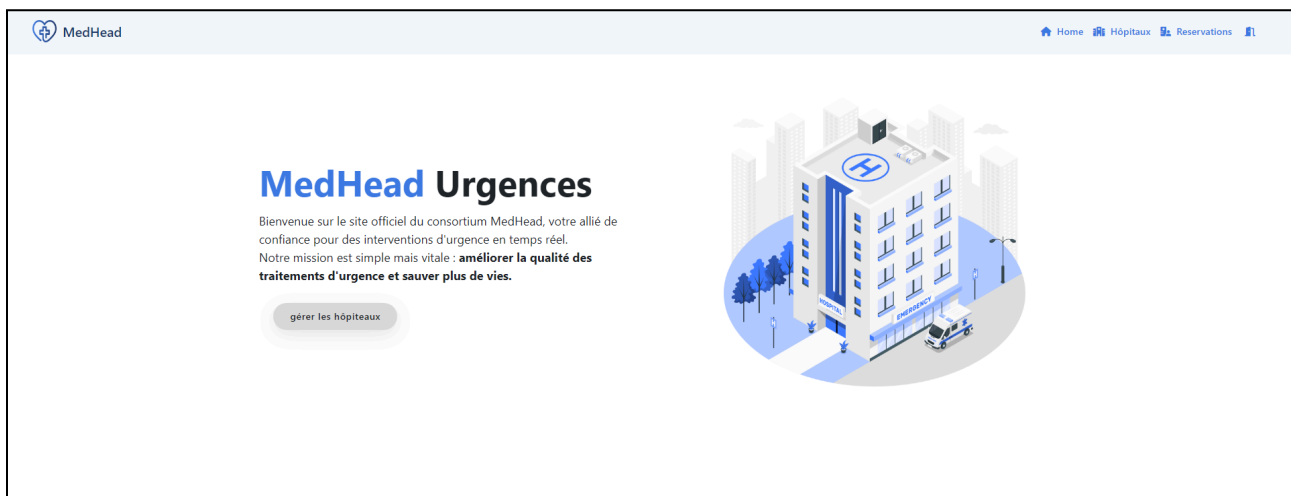


- L'email de confirmation, envoyé suite à la réservation, est accessible via l'interface Maildev, utilisée pour le développement. Nous pouvons récupérer cet email en accédant à <http://localhost:1080/#/>.



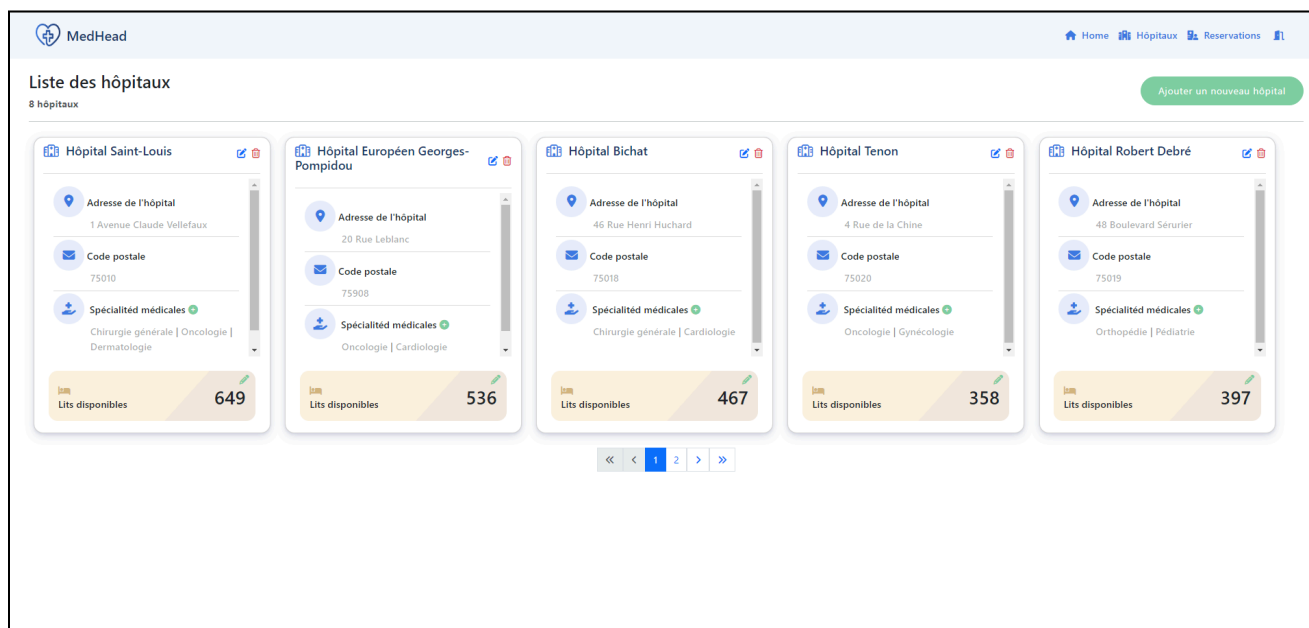
Page d'Accueil pour le l'Administrateur :

- Une fois connecté, l'administrateur est dirigé vers la page d'accueil. Cette page présente un résumé des principales fonctionnalités accessibles à l'administrateur, comme la gestion des réservations et des hôpitaux.



Gestion des Hôpitaux :

- L'administrateur peut voir la liste des hôpitaux disponibles dans la base de données, avec la possibilité gérer de hôpitaux, d'ajouter ou de modifier des spécialités (implémentation à venir).



Consultation des Réservations :

- L'administrateur peut consulter la liste des réservations effectuées par les patients, avec des options pour visualiser plus de détails ou effectuer des actions supplémentaires (implémentation à venir).

MedHead										Home	Hôpitaux	Réservations	
Liste des réservations													
4 réservations													
#	Reference réservation	Nom patient	Prénom patient	Numéro patient	Nom Organisation	Adresse hôpital	Code Postal hôpital	Lits disponibles	Spécialités	Actions			
1	ad351263-2f54-4310-8be6-0d5f53daa564	Admin	User	1234567890	Hôpital Necker	149 Rue de Sévres	75015	477	Spécialités				
2	5a65668e-55be-48d3-ac8e-64337e4aa6f4	Patient	User	0987654321	Hôpital Bichat	46 Rue Henri Huchard	75018	466	Spécialités				
3	c0fc50bd-eb61-4e3c-b972-c5adaa8bc66c	Patient	User	0987654321	Hôpital Européen Georges-Pompidou	20 Rue Leblanc	75908	535	Spécialités				
4	d486aa9e-2d47-443f-a117-e2f325860b76	Patient	User	0987654321	Hôpital Pitié-Salpêtrière	47-83 Boulevard de l'Hôpital	75013	975	Spécialités				

6.3. Documentation API Swagger

Documentation des Endpoints :








<http://localhost:8222/webjars/swagger-ui/index.html>

- Swagger génère une documentation complète des endpoints disponibles, comme illustré ci-dessous

Swagger		Select a definition
Open specification - Medhead Consortium		<ul style="list-style-type: none"> Hospital Management Service Hospital Management Service Reservation Service User Authentication Service hospitalManagement reservation userAuth
Servers		<ul style="list-style-type: none"> http://localhost:8222 - Local ENV
hospital-controller		
GET	/api/hospital/{hospital-id}	
PUT	/api/hospital/{hospital-id}	
DELETE	/api/hospital/{hospital-id}	
PUT	/api/hospital/updateBeds/{hospital-id}	
POST	/api/hospital/speciality	
POST	/api/hospital/addSpecialityToHospital	
POST	/api/hospital/addHospital	
GET	/api/hospital	
GET	/api/hospital/speciality?id={speciality-id}	
GET	/api/hospital/specialities	
GET	/api/hospital/nearest	

7. Conclusion et Recommandations

7.1. Évaluation de la Conformité à la Demande

Compétence	Critère d'évaluation	Évaluation	Statut
C.14. Conception d'une Architecture Adéquate	Le front-end et le back-end sont fonctionnels dans le respect des exigences.	Le front-end (Angular) et le back-end (Spring Boot) sont pleinement opérationnels, respectant les exigences de modularité et de scalabilité.	
		L'architecture microservices, incluant des composants comme Eureka et Spring Cloud Config, a été validée avec succès dans le cadre de ce PoC.	
C.16. Implémentation d'un Logiciel de Qualité	Les interactions entre le front-end et le back-end sont sécurisées.	Les interactions sont sécurisées grâce à l'utilisation de JWT , spring security pour l'authentification et de CORS pour contrôler les accès aux ressources backend.	
	Les tests vérifient la capacité du back-end à gérer une montée en charge.	Des tests de charge ont été effectués sur la fonctionnalité de recherche d'hôpitaux avec 1000 utilisateurs simultanés, confirmant que le back-end peut maintenir des performances optimales même sous une forte demande.	
C.17. Tests Logiciels à Plusieurs Niveaux	Un pipeline CI/CD est intégré au repository du code source.	Un pipeline CI/CD a été mis en place via GitHub Actions, automatisant les tests unitaires, d'intégration, et E2E, ainsi que la compilation des microservices et du front-end.	
	Des tests vérifient le fonctionnement.	Les tests effectués couvrent les principales fonctionnalités, assurant la conformité aux spécifications et la non-régression des fonctionnalités développées.	
C.18. Application d'Analyse de Données Massives	Les résultats de la PoC fournissent des métriques quant au temps d'exécution de l'API.	Les tests de performance ont permis de collecter des métriques précises sur les temps de réponse des API, fournissant une base pour l'optimisation future des performances.	

7.2. Recommandations pour la Mise en Œuvre de la Solution Finale

En fonction des résultats obtenus, voici quelques recommandations pour la suite du projet :

Finalisation et Extension des Fonctionnalités :

- **Compléter les fonctionnalités administratives :**
Terminer le développement des fonctionnalités CRUD pour la gestion des hôpitaux et des spécialités, ainsi que d'autres actions administratives.
- **Amélioration de l'expérience utilisateur**

Renforcement des Tests :

- **Étendre la couverture des tests côté backend et frontend :**
 - Il est essentiel de renforcer les tests unitaires et d'intégration pour couvrir tous les microservices, garantissant ainsi la robustesse du système dans son ensemble.
 - Ajouter des tests unitaires côté frontend et compléter les tests E2E.
- **Optimiser les performances :**
Continuer à affiner les performances des API, en particulier celles impliquées dans des scénarios à haute charge, comme la recherche d'hôpitaux et la réservation des lits.

Intégration du Déploiement Automatisé :

- **Automatisation du déploiement :**
Intégrer des étapes de déploiement automatisé dans les pipelines CI/CD pour faciliter les déploiements en production et minimiser les erreurs humaines.

Sécurité et Conformité :

- **Réaliser un audit de sécurité :**
Assurer la conformité avec les réglementations en matière de protection des données, telles que le RGPD avant le déploiement en production.

Ces recommandations ne sont pas définitives et devront être ajustées en fonction des exigences fonctionnelles et non fonctionnelles spécifiées pour la solution finale.

7.3. Conclusion

Ce Proof of Concept (PoC) a validé l'architecture technique et les choix technologiques envisagés pour la plateforme de gestion des réservations de lits d'hôpitaux. Le système, basé sur une architecture microservices avec Spring Boot et Angular, a démontré sa capacité à répondre aux exigences fonctionnelles.

En résumé, ce PoC fournit une base solide pour le développement futur de la plateforme, avec des recommandations pour la transition vers une solution complète et prête pour la production.