REPORTING POC MEDHEAD

MEDHEAD

TABLE DES MATIERES

OBJECTIFS ET EXIGENCES DE LA POC

OBJECTIFS

- Améliorer la qualité des traitements d'urgences et de sauver plus de vie
- Gagner la confiance des utilisateurs quant à la simplicité d'un tel système
- 90% des cas d'urgences sont acheminés vers l'hôpital compétent le plus proche
- Temps moyen de traitement d'une urgence passe de 18,25 minutes (valeur actuelle) à 12 minutes (valeur souhaitée)
- Temps de réponse de moins de 200 millisecondes avec une charge de travail allant jusqu'à 800 requêtes par seconde, par instance de service
- La mise en œuvre respecte les normes imposées et est terminée dans le délai impart

EXIGENCES

Exigence	Détails
API RESTful	Fournir une API RESTful qui renvoie le lieu où se rendre, **basée sur la technologie Java (imposée par le consortium)** et pouvant s'inscrire dans une architecture microservice.
Interface graphique	Une simple page permettant de sélectionner une spécialité et de saisir la localisation. Le framework imposé est Angular, React, ou VueJS.
Protection des données	S'assurer que toutes les données du patient sont correctement protégées.
Validation avec tests	La PoC doit être entièrement validée avec des tests reflétant la pyramide des tests : unitaires, d'intégration, et E2E. Des tests de stress doivent être inclus pour garantir la continuité d'activité en cas de pic d'utilisation.
Intégration dans le futur développement	S'assurer que la PoC est facilement intégrable dans le développement futur et que le code est partageable. Un pipeline CI/CD doit être fourni.
Bloc de construction pour l'équipe	S'assurer que les équipes de développement peuvent utiliser la PoC comme base pour d'autres modules ou comme modèle à suivre.
Versionnage du code	Le code doit être versionné à l'aide d'un workflow Git adapté.

Documentation technique	Un fichier readme.md doit être fourni avec des instructions pour l'exécution des tests, le fonctionnement du pipeline, et le workflow Git utilisé.
Document de reporting	Un document de reporting doit être rédigé, justifiant les technologies, le respect des normes (ex. RGPD, architecture microservices), et présentant les résultats et enseignements de la PoC.

STACK TECHNIQUE

BACK-END

La technologie Java étant imposé pour le back-office, il a été décidé d'utiliser le framework Spring Boot.

L'utilisation de Spring Boot pour développer l'API RESTful dans ce projet se justifie par plusieurs avantages et caractéristiques qui répondent aux exigences techniques et aux bonnes pratiques de développement logiciel :

Facilité de mise en place et gain de temps

- Spring Boot permet de configurer rapidement une application grâce à son approche d'auto-configuration et à des conventions intelligentes. Cela permet de créer une API RESTful complète en très peu de temps, tout en bénéficiant de l'écosystème riche de Spring.
- Il élimine la plupart des tâches de configuration manuelles, ce qui accélère le développement, surtout dans le cadre d'une preuve de concept (PoC).

Architecture Microservices

- Spring Boot est parfaitement adapté aux architectures microservices, qui sont de plus en plus courantes dans les systèmes distribués modernes.
- La scalabilité et la modularité offertes par Spring Boot permettent de déployer facilement de multiples instances d'une API, réparties sur différentes infrastructures, tout en garantissant une communication fluide entre les services

Sécurité des données

- Spring Security, un sous-module de Spring Boot, est un puissant outil permettant de mettre en place des mesures de sécurité robustes. Il permet l'authentification, l'autorisation et la protection des données sensibles (comme les informations des patients) via des standards comme OAuth2 et JWT.
- Ces fonctionnalités sont essentielles pour assurer le respect des réglementations telles que la RGPD (Règlement Général sur la Protection des Données).

Facilité de test

- Spring Boot est conçu pour faciliter les tests. Grâce à des outils intégrés comme Spring Boot Test, il est possible de tester les contrôleurs, les services et les repositories de manière unitaire et d'intégration avec des dépendances simulées (mocking).
- Les tests d'intégration, nécessaires pour valider une architecture microservice, sont simplifiés grâce à l'auto-configuration des environnements de test.

FRONT-END

Le front-end s'appuiera sur le framework Angular.

Angular est un framework de développement front-end populaire et robuste qui facilite la création d'applications web modernes et réactives.

Il est basé sur TypeScript, ce qui apporte des avantages comme la détection d'erreurs à la compilation et une meilleure maintenabilité du code. Angular suit une architecture composant-service, ce qui permet de structurer efficacement le code, facilitant la réutilisation et l'évolution des fonctionnalités.

Grâce à ses outils intégrés pour les tests unitaires et d'intégration (via Jasmine et Karma) et son support natif pour les Single Page Applications (SPA), Angular garantit un développement rapide et une excellente expérience utilisateur.

BASE DE DONNEES

Les bases de données seront fournies par PostgreSQL.

Fiabilité et transactions ACID : PostgreSQL est une base de données relationnelle robuste, offrant des transactions ACID pour garantir l'intégrité des données, essentielles pour les informations sensibles liées aux hôpitaux et spécialités.

Conformité et sécurité : Il propose des fonctionnalités de sécurité avancées, telles que le chiffrement SSL, la gestion des rôles, et le contrôle d'accès, répondant aux exigences de protection des données (notamment RGPD).

Scalabilité et performance : PostgreSQL est capable de supporter une forte charge de travail avec des options de scalabilité (réplication synchrone/asynchrone), garantissant des performances stables même en cas de pic d'utilisation.

CI/CD

La chaîne CI/CD sera configuré grâce à Jenkins.

Jenkins est un outil open-source largement utilisé pour automatiser les étapes du cycle de développement logiciel.

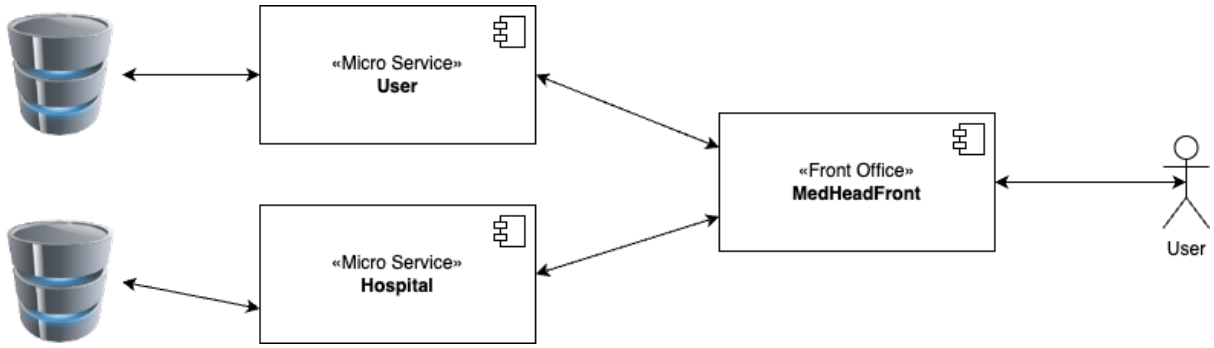
Il permet de déclencher des builds, des tests, et des déploiements automatiquement à chaque changement de code, garantissant ainsi une intégration continue et une livraison rapide.

Jenkins est hautement **extensible** grâce à ses nombreux plugins, permettant d'intégrer facilement divers outils de développement, de tests et de déploiement (comme Docker, Cypress, SonarQube, etc.).

Sa **flexibilité** et sa large adoption en font un choix idéal pour gérer des pipelines CI/CD efficaces et fiables.

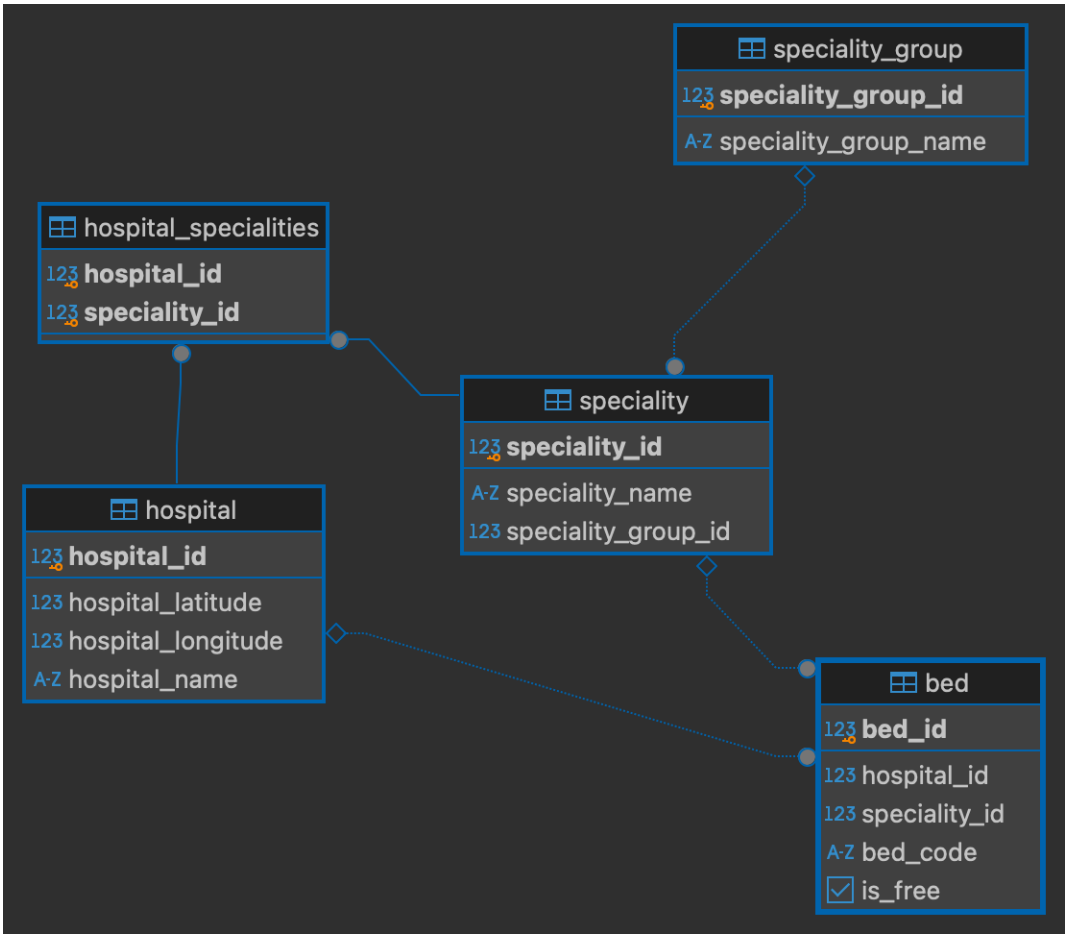
ARCHITECTURE

TECHNIQUE

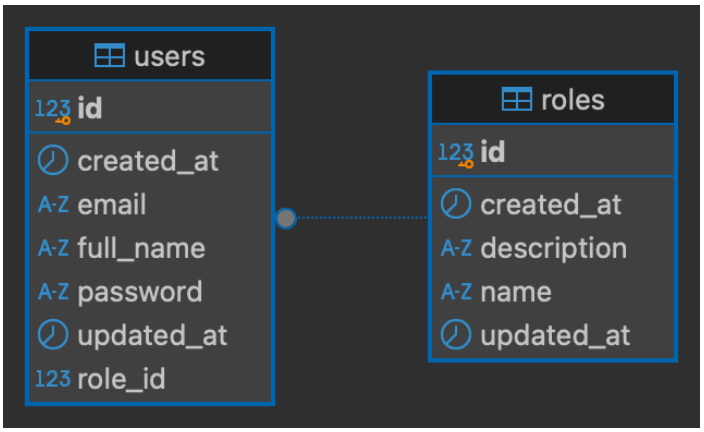


MPD

HOSPITAL



USERS



RESPECT DES NORMES

CONFORMITE AU RGPD (REGLEMENT GENERAL SUR LA PROTECTION DES DONNEES)

La solution respecte les exigences du RGPD en matière de protection des données personnelles des patients. Voici comment nous avons assuré cette conformité :

Collecte minimale de données : Seules les informations nécessaires au traitement des requêtes des patients (comme la localisation et les spécialités médicales) sont collectées.

Stockage sécurisé des données : Toutes les données sensibles, comme les informations des patients, sont chiffrées à la fois au repos (dans la base de données PostgreSQL) et en transit (via HTTPS). De plus, les tokens JWT (JSON Web Token) utilisés pour l'authentification sont sécurisés et stockés de manière appropriée.

RESPECT DES PRINCIPES D'ARCHITECTURE MICROSERVICES

L'architecture microservices a été choisie pour la PoC, conformément aux principes de modularité, de scalabilité, et de découplage :

Modularité et découplage : Chaque fonctionnalité est implémentée dans un service distinct. Cela permet une maintenance facilitée et un déploiement indépendant de chaque module.

Évolutivité : L'API RESTful est conçue pour être distribuée sur plusieurs instances afin de gérer des pics de charge, ce qui améliore la scalabilité horizontale du système.

Communication entre services : Les services communiquent via des API REST, facilitant l'intégration d'autres modules ou microservices dans le futur.

SECURITE ET PROTECTION DES ACCES

Authentification et autorisation : L'application utilise des tokens JWT pour sécuriser les échanges avec l'API. Chaque requête est validée par une authentification préalable, ce qui garantit que seules les personnes autorisées ont accès aux données.

Hachage des mots de passe : Les mots de passe utilisateurs sont stockés de manière sécurisée en utilisant un algorithme de hachage robuste, tel que bcrypt, pour prévenir toute compromission.

RESPECT DES PRINCIPES DE CONTINUITE D'ACTIVITE

Afin de garantir la continuité des services, la PoC a été éprouvée avec des tests de stress. Ces tests simulent des pics de charge pour évaluer les performances du système en situation d'usage intensif :

Tests de charge : Nous avons effectué des tests de charge avec l'outil Postman pour simuler un grand nombre d'utilisateurs simultanés.

Résilience : En cas de surcharge ou de panne d'un microservice, l'architecture permettra de rediriger les requêtes vers d'autres instances, garantissant ainsi la résilience du système.

NORMES DE DEVELOPPEMENT ET BONNES PRATIQUES

CI/CD (Intégration continue et livraison continue) : Le projet utilise Jenkins pour l'automatisation des tests et du build, assurant un processus de livraison rapide et fiable. Chaque modification du code déclenche les tests.

Versionnement du code : Le projet suit un workflow Git adapté (Gitflow) avec des branches dédiées pour le développement, les features, et la production. Cela garantit une gestion structurée des versions et un suivi des changements.

RESULTATS ET ENSEIGNEMENT DE LA POC

TESTS

TESTS UNITAIRES

MICRO SERVICE HOSPITAL

ID de test	Description du test	Composant testé
1	Vérifie que la méthode findFreeBedsBySpecialityId renvoie une liste de lits disponibles pour une spécialité donnée.	BedServiceImpl
2	Vérifie que la méthode changeBedState bascule l'état de disponibilité d'un lit et renvoie le lit mis à jour.	BedServiceImpl
3	Vérifie que la méthode bulkSave crée et enregistre plusieurs lits dans la base de données.	BedServiceImpl
4	Vérifie que la méthode bulkSave lève une exception lorsque l'hôpital n'est pas trouvé.	BedServiceImpl
5	Vérifie que la méthode bulkSave lève une exception lorsque la spécialité n'est pas trouvée.	BedServiceImpl

6	Vérifie que la méthode extractInt renvoie un entier correctement extrait d'un code de lit.	BedServiceImpl
7	Vérifie que la méthode extractInt lève une exception lorsque le format du code de lit est invalide.	BedServiceImpl
8	Vérifie que la méthode findAll renvoie une liste de tous les hôpitaux.	HospitalServiceImpl
9	Vérifie que la méthode findById renvoie un hôpital existant lorsque l'ID est trouvé.	HospitalServiceImpl
10	Vérifie que la méthode findById renvoie un Optional vide lorsque l'ID de l'hôpital n'est pas trouvé.	HospitalServiceImpl
11	Vérifie que la méthode findHospitalWithFreeBedsForOneSpeciality renvoie les hôpitaux avec des lits libres pour une spécialité donnée.	HospitalServiceImpl
12	Vérifie que la méthode findAll renvoie une liste de tous les groupes de spécialités.	SpecialityGroupServiceImpl

MICRO SERVICE USER

ID de test	Description du test	Composant testé
13	Devrait récupérer le token depuis le sessionStorage	JwtService
14	Devrait stocker le token dans le sessionStorage	JwtService
15	Devrait supprimer le token du sessionStorage	JwtService
16	Devrait récupérer un utilisateur par ID	UserService
17	Devrait créer un nouvel utilisateur	UserService
18	Devrait mettre à jour un utilisateur existant	UserService
19	Devrait supprimer un utilisateur par ID	UserService

TESTS D'INTEGRATIONS

MICRO SERVICE HOSPITAL

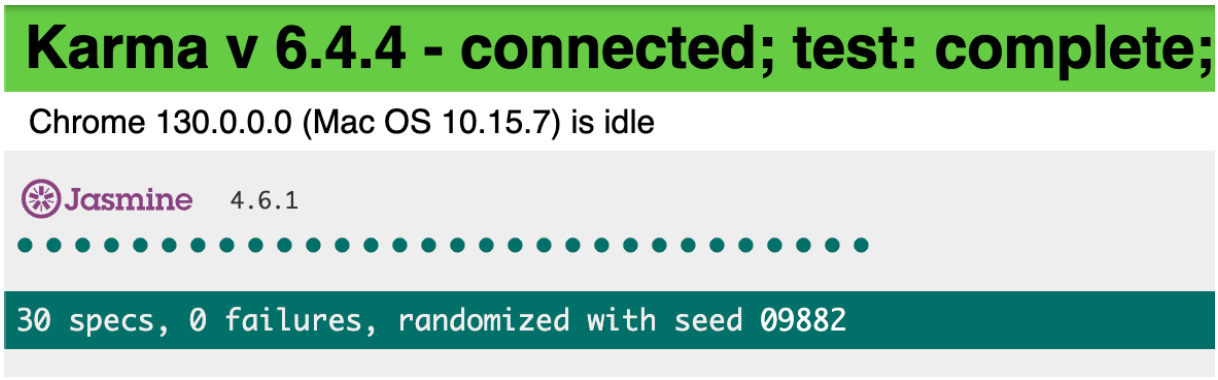
ID de test	Description du test	Composant testé
20	Vérifie la récupération des lits disponibles via un point d'entrée GET.	BedController
21	Teste la réservation d'un lit spécifique via un point d'entrée POST.	BedController
22	Vérifie la récupération des hôpitaux proches en fonction de la latitude, longitude et spécialité.	HospitalController
23	Confirme la récupération des groupes de spécialité via un point d'entrée GET.	SpecialityGroupController
24	Teste la création d'un nouveau groupe de spécialité via un point d'entrée POST.	SpecialityGroupController

MICRO SERVICE USER

ID de test	Description du test	Composant testé
25	Vérifie la fonctionnalité de connexion de l'administrateur.	AdminController
26	Vérifie si l'administrateur peut ajouter un utilisateur.	AdminController
27	Confirme que l'administrateur peut supprimer un utilisateur.	AdminController
28	S'assure qu'un jeton est généré lors de l'authentification.	AuthenticationController
29	Valide la révocation du jeton pour un utilisateur spécifique.	AuthenticationController
30	Teste le point d'entrée pour l'enregistrement d'un utilisateur.	UserController
31	Vérifie que les informations de l'utilisateur sont retournées avec précision.	UserController

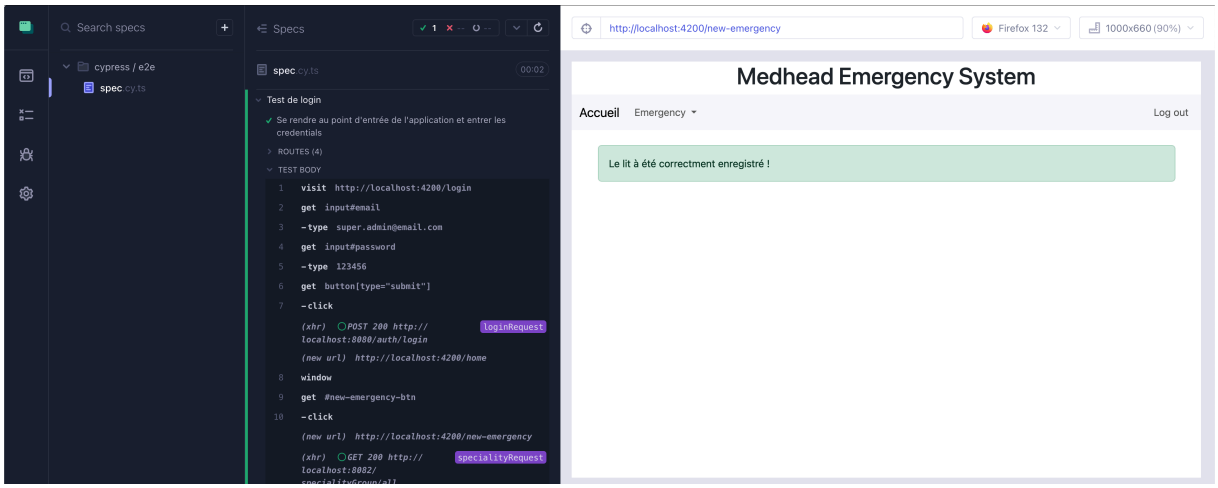
32	S'assure que la fonctionnalité de réinitialisation de mot de passe fonctionne correctement.	UserController
33	Teste la récupération de tous les utilisateurs par l'administrateur.	AdminController
34	Confirme qu'un jeton valide est nécessaire pour accéder aux points d'entrée sécurisés des utilisateurs.	AuthenticationController

TESTS KARMA



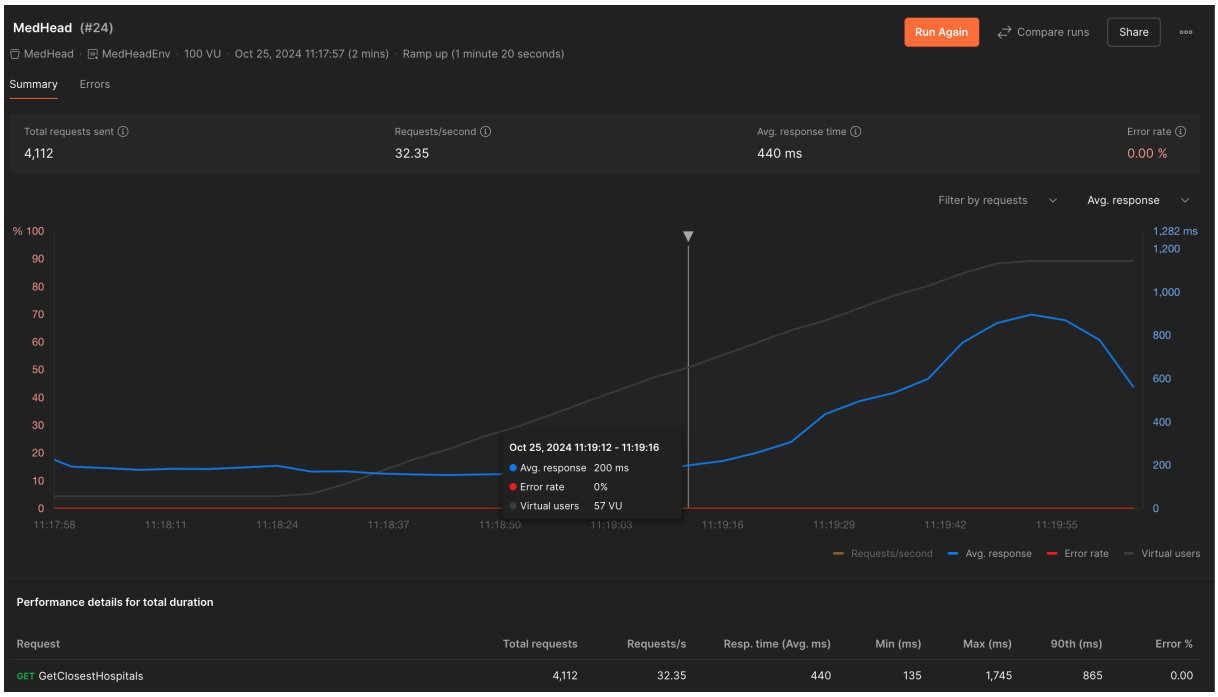
Tous les tests Jasmine ont réussi avec 30 spécifications exécutées sans aucune erreur. Cela reflète une validation complète et fiable des fonctionnalités testées. Le seed aléatoire utilisé garantit également la robustesse face à différents ordres d'exécution.

TEST E2E



Le test Cypress a été exécuté avec succès, validant la connexion et l'enregistrement correct d'une nouvelle urgence dans l'application. L'interface utilisateur confirme également que l'action a été complétée sans erreur.

TEST DE CHARGE

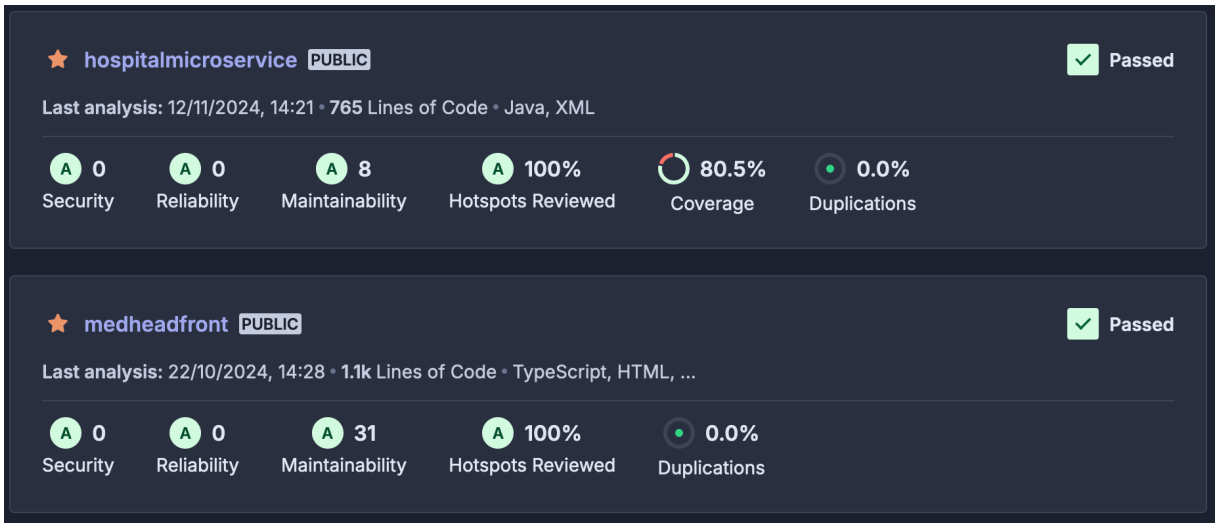


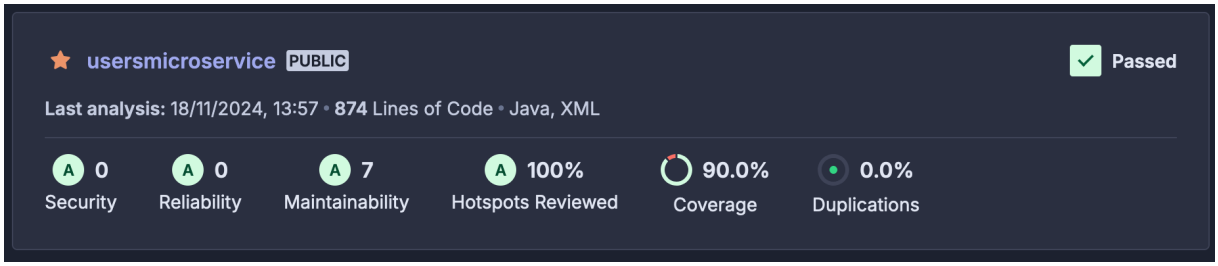
Ce test de montée en charge, réalisé en local sur un poste personnel, a permis d'évaluer les performances de l'endpoint GetClosestHospitals dans un environnement contrôlé, mais non représentatif d'une situation réelle en production. Avec 100 utilisateurs virtuels simulés et un total de 4 112 requêtes envoyées sur une période de deux minutes, le système a maintenu un temps de réponse moyen de 440 ms, sans aucune erreur enregistrée.

Cependant, les résultats obtenus doivent être interprétés avec **prudence**, car l'exécution en local impose des limitations importantes, telles que les capacités réseau, les ressources matérielles et l'absence de latence typique des environnements distribués. Par conséquent, ces résultats ne peuvent pas refléter les performances réelles de l'application en production.

Pour une évaluation plus précise, il serait nécessaire d'effectuer des tests similaires dans un environnement de pré-production, avec une infrastructure iso à celle de production, afin de simuler des conditions réalistes et de mieux comprendre la capacité réelle du système à gérer des charges importantes. Ces tests permettront également de détecter d'éventuels goulets d'étranglement ou problèmes de scalabilité dans des conditions proches de l'utilisation finale.

COUVERTURE DU CODE ET AUDIT





Ces résultats reflètent une **attention particulière à la qualité logicielle** et aux **bonnes pratiques de développement** dans les différents services analysés. L'absence de failles de sécurité, de duplications, et les scores élevés de maintenabilité et de couverture montrent que le projet est sur une base solide, tant pour l'évolutivité que pour la fiabilité.

Pour aller plus loin, il serait intéressant de viser une couverture de 100 % pour les microservices afin d'assurer une robustesse maximale et de maintenir cet excellent niveau de qualité dans les futurs développements.