

# Document de reporting



Entreprise : Medhead

Nom du projet : Faites adhérer les parties prenantes avec un POC

Date de commencement du projet : 09/2024

# TABLER DES MATIÈRES

## Introduction

Objet du document  
Contexte

## Justification des technologies

Front-end  
Back-end  
Authentification  
API de recherche d'adresse  
Base de données  
Hébergement  
Gestion des images

## Conformité aux exigences de la Proof of Concept

## CI/CD et automatisation

## Tests

Tests unitaires  
Tests de charge  
Tests E2E  
Tests d'intégration  
Set de données

## Sécurité

Token JWT  
HTTPS  
Cryptage

## Résultats de la Proof of Concept

## Points de surveillance

## Tester l'application

Back-end  
Front-end

## Liens utiles

## Conclusion

# Introduction

## Objet du document

Le document de reporting permet de justifier les résultats de la POC et de prouver sa conformité avec les éléments détaillés dans le document des exigences de la POC.

Ce document montre que les normes et principes sont respectés et que le système fonctionne correctement et respecte le RGPD.

## Contexte

MedHead Consortium est un regroupement de grandes institutions médicales anglaises. Le Consortium souhaite créer un système pour pallier aux risques liés au traitement des recommandations de lits d'hôpitaux dans des situations d'intervention d'urgence.

La preuve de concept (POC) doit montrer l'efficacité de l'architecture et des technologies choisies pour la nouvelle plateforme.



## Justification des technologies

### Front-end

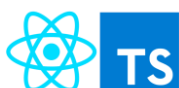
La technologie choisie pour le front-end est ReactJS / typescript.

**ReactJS** est une technologie moderne possédant une large communauté. De plus, il y a des mises à jour régulières et il existe des technologies similaires pour développer une solution mobile (react native). Il existe de nombreuses bibliothèques qui facilitent le développement.

**Typescript** permet quant à lui d'éviter des bugs lors du développement en ajoutant le typage statique ce qui rends par exemple le refactoring plus simple car les développeurs seront informés des erreurs de typage. De plus, il est compatible avec les bibliothèques javascript.

### Back-end

La technologie utilisée est **java**. Cette technologie est imposée par le Consortium.



La technologie **Maven** a également été choisie pour la gestion des dépendances. Cette gestion est simplifiée par l'outil grâce au fichier pom.xml. De plus, Maven permet également d'automatiser les builds (compile, build, test, install, ...).

**Springboot** a également été sélectionné pour générer le projet par défaut via l'outil en ligne Spring Initializr (<https://start.spring.io/>). Les plugins et dépendances de springboot facilite le développement ; notamment avec spring security, ou d'autres librairies permettant de lier la base de données au back-end facilement.



## Authentication

Afin de sécuriser la plateforme, une authentification a été mise en place.

L'API possède un endpoint (point d'entrée) permettant d'effectuer une requête de connexion qui va générer un **token JWT** si les informations d'identification sont correctes.

Utiliser les tokens JWT permet de **sécuriser les requêtes**, en les rendant accessibles aux utilisateurs connectés possédant le rôle correct.

Le token JWT possède une durée de validité, qui a été configurée à 1 jour dans le système.

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjMONTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c		
Header	Payload	Signature
Type de token et algorithme (SHA256)	Informations de l'utilisateur, expiration du token	Résultat de la signature du header et de la payload avec une clé secrète

*Contenu d'un token*

## API de recherche de point de coordonnées géographique

L'api **Nominatim Openstreetmap** a été utilisée dans le composant front-end.

Cette API est gratuite et permet d'effectuer une recherche d'adresse en fonction de divers critères, notamment le nom de l'adresse.

Dans le cadre de la POC, nous avons fixé certaines propriétés telle que le pays (France) et le nombre de résultats affichés (5), afin de faciliter la recherche. Ces propriétés sont facilement modifiables.

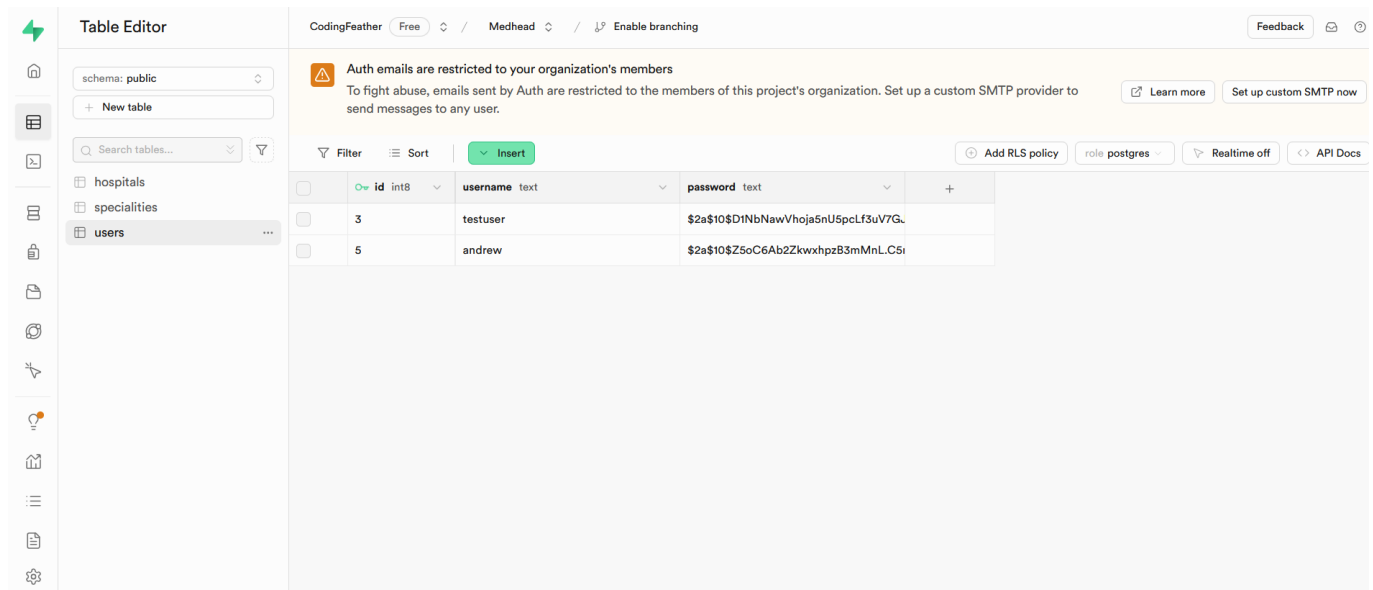
L'utilisation de cette API permet de récupérer les coordonnées géographiques (latitude et longitude) exactes d'une adresse.



**OpenStreetMap**

## Base de données

La base de données utilisée est une base de données PostgreSQL, disponible sur l'outil gratuit **Supabase**.

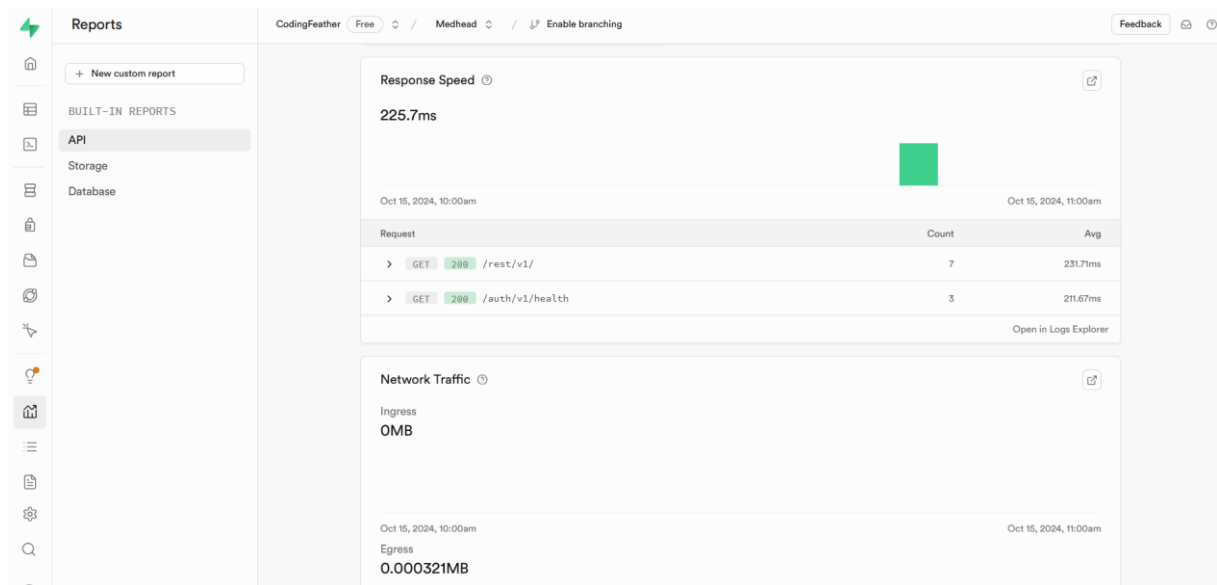


### Interface de l'outil Supabase

Supabase propose une interface visuelle pour gérer simplement la base de données ainsi qu'une interface pour faire des actions sur celles-ci via des commandes.

L'outil propose une documentation détaillée sur la façon dont nous pouvons intégrer cette base de données à notre composant front-end / back-end et propose d'autres fonctionnalités supplémentaires au-delà de la base de données (ex : gestion des utilisateurs / authentification).

Enfin, l'outil propose également une interface des rapports générés (informations sur la base de données, nombres de connexion etc.).



### Rapport Supabase

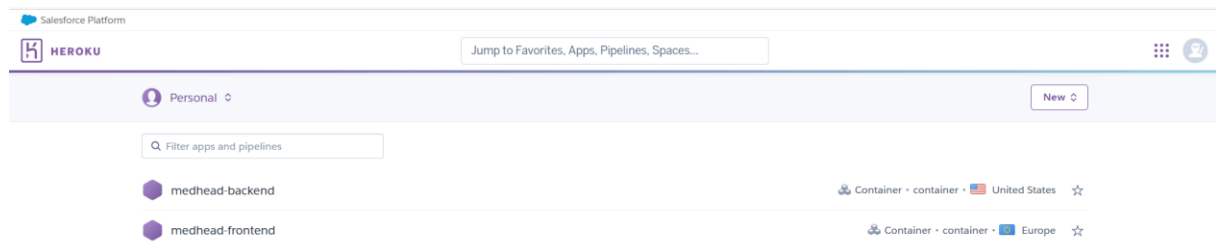
## Hébergement

Un outil d'hébergement a été choisi ; **Heroku**, plus précisément son plan de base.

Heroku est une solution payante qui permet d'héberger des applications (appelées conteneurs).

Par défaut, Heroku attribue une URL publique unique à chaque conteneur mais il est possible de configurer l'URL pour utiliser son propre nom de domaine.

Dans le cadre du projet Medhead, il y a deux conteneurs ; un pour le front-end et un pour le back-end.



### *Interface d'Heroku*

L'avantage d'Heroku est sa simplicité d'utilisation, il est facile de lier les conteneurs à des outils externes.

Dans notre cas, nous utilisons un hook pour lier Heroku à Dockerhub afin de publier automatiquement les nouvelles versions.

De plus, il existe divers services supplémentaires, comme un service pour créer et gérer ses bases de données. Cependant, contrairement à Supabase, l'ajout et le maintien d'une base de données Heroku est payant, en plus de la formule de base Heroku.

## **Gestion des images**

La pipeline CI/CD génère et pousse des images Docker automatiquement vers **Dockerhub**.

Un *hook* Dockerhub a été configurée, qui vient ensuite pousser la nouvelle image automatiquement sur Heroku, pour la partie front-end, comme expliqué sur le schéma ci-dessus.

Dockerhub est une solution très populaire pour la conteneurisation et possède beaucoup d'images officielles.

A noter, il est nécessaire d'avoir Docker (Docker Desktop sous Windows) afin de déployer le back-end via les commandes.



# **Conformité aux exigences de la Proof of Concept**

Cette section reprend une partie du document des exigences de la POC.

Les exigences suivantes ont été convenues lors de la définition de cette hypothèse :

- Fournir une API RESTful qui renvoie le lieu où se rendre :
  - La technologie Java est imposée par le consortium,
  - L'API doit pouvoir s'inscrire à terme dans une architecture microservice ;
- Fournir une interface graphique qui consomme l'API :
  - Une simple page permettant de sélectionner une spécialité et de saisir la localisation est suffisante,
  - Le consortium impose d'utiliser l'un des frameworks Javascript/Typescript courant du marché : Angular, React, VueJS ;
- S'assurer que toutes les données du patient sont correctement protégées ;
- S'assurer que votre PoC est entièrement validée avec des tests reflétant la pyramide de tests (tests unitaires, d'intégration et E2E) :
  - L'API doit être éprouvée avec des tests de stress pour garantir la continuité de l'activité en cas de pic d'utilisation ;
- S'assurer que la PoC peut être facilement intégrée dans le développement futur : rendre le code facilement partageable, fournir des pipelines d'intégration et de livraison continue (CI/CD) ;
- S'assurer que les équipes de développement chargées de cette PoC sont en mesure de l'utiliser comme un bloc de construction pour d'autres modules ou tout du moins comme un modèle à suivre ;
- Le code est versionné à l'aide d'un workflow Git adapté ;
- La documentation technique de la PoC sera formalisée dans un fichier readme.md respectant le format markdown et contiendra au minimum :
  - les instructions pour l'exécution des tests,
  - le fonctionnement du pipeline,
  - le workflow Git retenu (ce dernier sera détaillé pour qu'il soit réutilisable par les équipes) ;
- Un document de reporting sera rédigé indiquant :
  - une justification des technologies utilisées,
  - une justification du respect des normes et des principes (exemple : norme RGPD, principe d'architecture microservice, etc.),
  - les résultats et les enseignements de la PoC

# CI/CD et automatisation

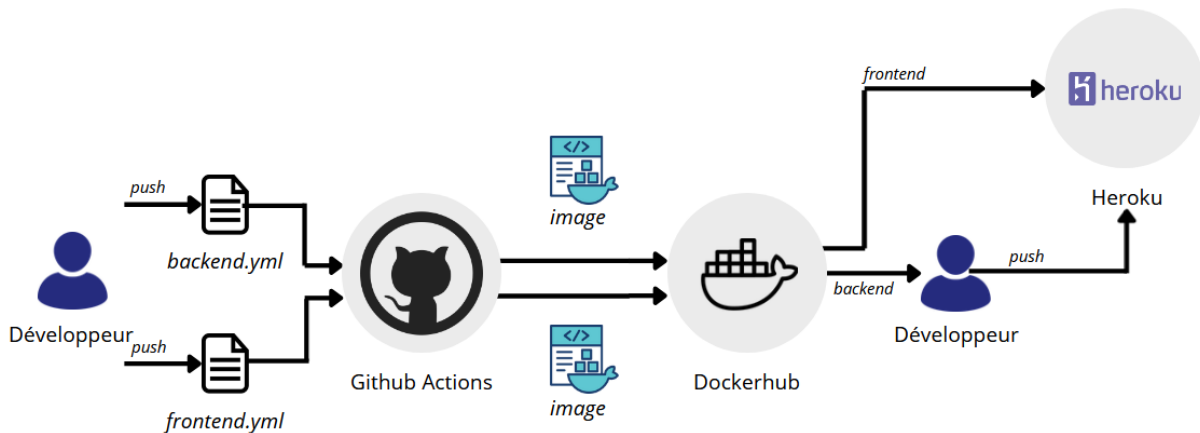
Nous utilisons des fichiers workflows pour gérer l'automatisation.

Actuellement, ils sont configurés pour déclencher la CI/CD **Github Actions** lorsqu'une modification est apportée sur le backend ou le frontend sur la branche principale (main).

Github actions va exécuter les commandes du workflow ; le build, test, push, etc.

Le workflow construit et déploie une nouvelle image docker sur **Dockerhub**.

Ensuite, le hook liant le conteneur dockerhub du **front-end** permet de déployer automatiquement la nouvelle version sur Heroku. Cependant, il faut exécuter certaines commandes à la main pour déployer le **back-end** sur Heroku due à une restriction de Heroku sur l'hébergement d'API.



*Automatisation du déploiement*

Les workflows fonctionnent ainsi :

## **Back-end :**

- Se déclenche lors d'un push sur main affectant le dossier backend.
- Utilise java JDK 22.
- Effectue les tests unitaires et tests d'intégration avec maven.
- Génère l'image Docker.
- Connexion à Dockerhub
- Publie l'image sur Dockerhub si tous les tests réussissent.

## **Front-end :**

- Se déclenche lors d'un push affectant le dossier frontend sur la branche main.
- Créer l'image Docker
- Connexion à Dockerhub
- Pousse l'image sur Dockerhub
- Connexion à Heroku
- Déploiement sur Heroku



# Tests

## Tests unitaires

Les tests unitaires permettent de tester des éléments isolés ; le fonctionnement d'une petite partie du code, d'un endpoint.

Les tests unitaires suivants ont été créés et ajoutés à la CI/CD pour l'API :

- **Authentification** : 2 tests ; teste le cas où un utilisateur s'authentifie correctement et le cas où les identifiants sont incorrects.
- **Hôpitaux** : 4 tests ; teste la récupération des hôpitaux proches, la récupération des listes d'hôpitaux, et la réservation de lit.
- **Spécialités** : 2 tests ; teste la récupération de toutes les spécialités et teste la récupération d'une spécialité spécifique.

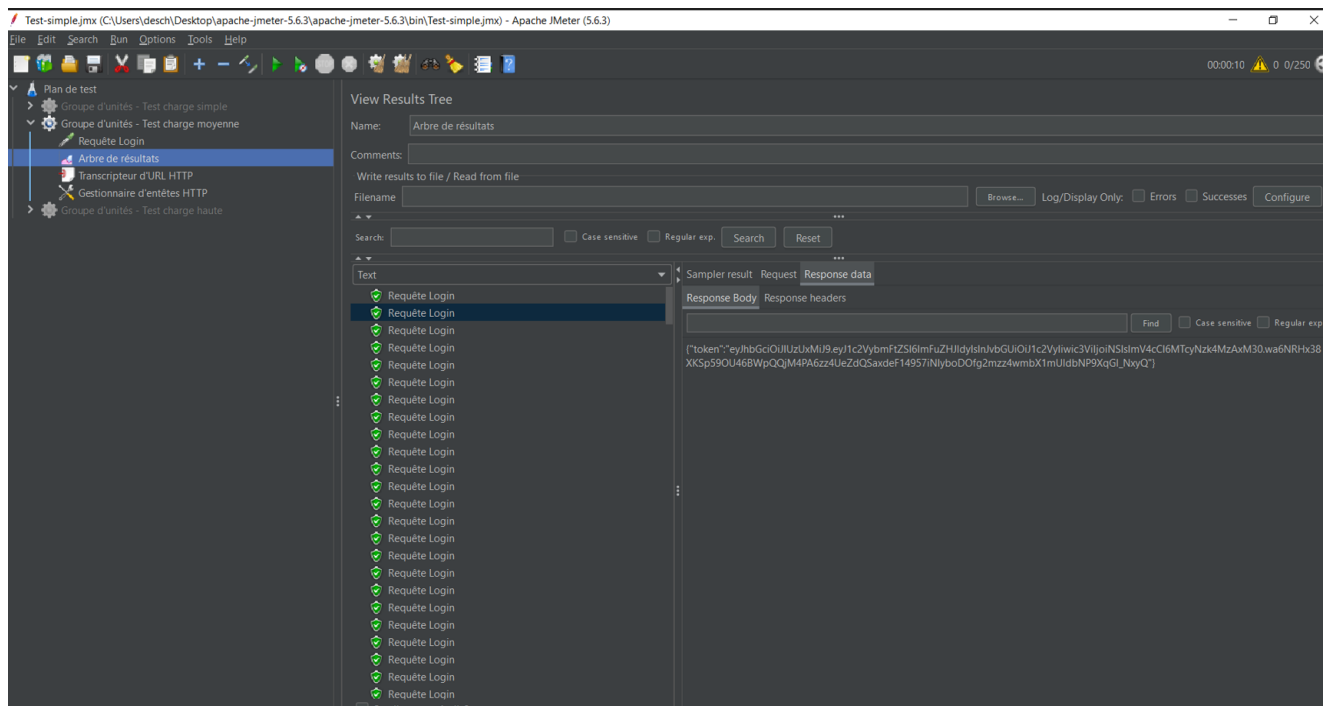
## Tests de charge

Les tests de charge permettent d'évaluer la performance de l'API et d'en détecter les limites.

Pour effectuer ces tests, nous avons utilisés JMeter. Les trois plans de charge utilisés ont été exportés et sont disponibles dans le projet Devops-POC contenant le code.

Fonctionnement :

1. Installer JMeter ([https://jmeter.apache.org/download\\_jmeter.cgi](https://jmeter.apache.org/download_jmeter.cgi))
2. Exécuter le JAR local.
3. Importer le plan de test disponible dans le code source *jmeter/JMETER\_Stress\_Test\_Plan.jmx*
4. Cliquer sur le bouton Lancer et sélectionner l'arbre de résultat.



Résultats du test de charge moyenne

Configurations des tests de charge :

#### Test de charge simple

- Utilisateurs : 10
- Montée en charge : 10
- Nombre d'itérations : 1

Résultats : Taux de réussite de 100%

#### Test de charge moyenne

- Utilisateurs : 250
- Montée en charge : 10
- Nombre d'itérations : 1

Résultats : Taux de réussite de 100%

#### Test de charge haute

- Utilisateurs : 950
  - Montée en charge : 10
  - Nombre d'itérations : 1
- Résultats : Taux d'échec de 50% environ

### **Tests d'intégration**

Les tests d'intégration permettent de vérifier le bon fonctionnement des interactions entre les différents composants du système.

Les tests suivants ont été créés et ajoutés à la CI/CD pour l'API :

- **Hôpitaux** : 3 tests ; récupération de deux hôpitaux proches d'un point de coordonnées spécifique, réservation d'un lit d'hôpital, échec d'une réservation d'hôpital.
- **Spécialité** : 1 test ; récupération d'une spécialité.

### **Tests E2E**

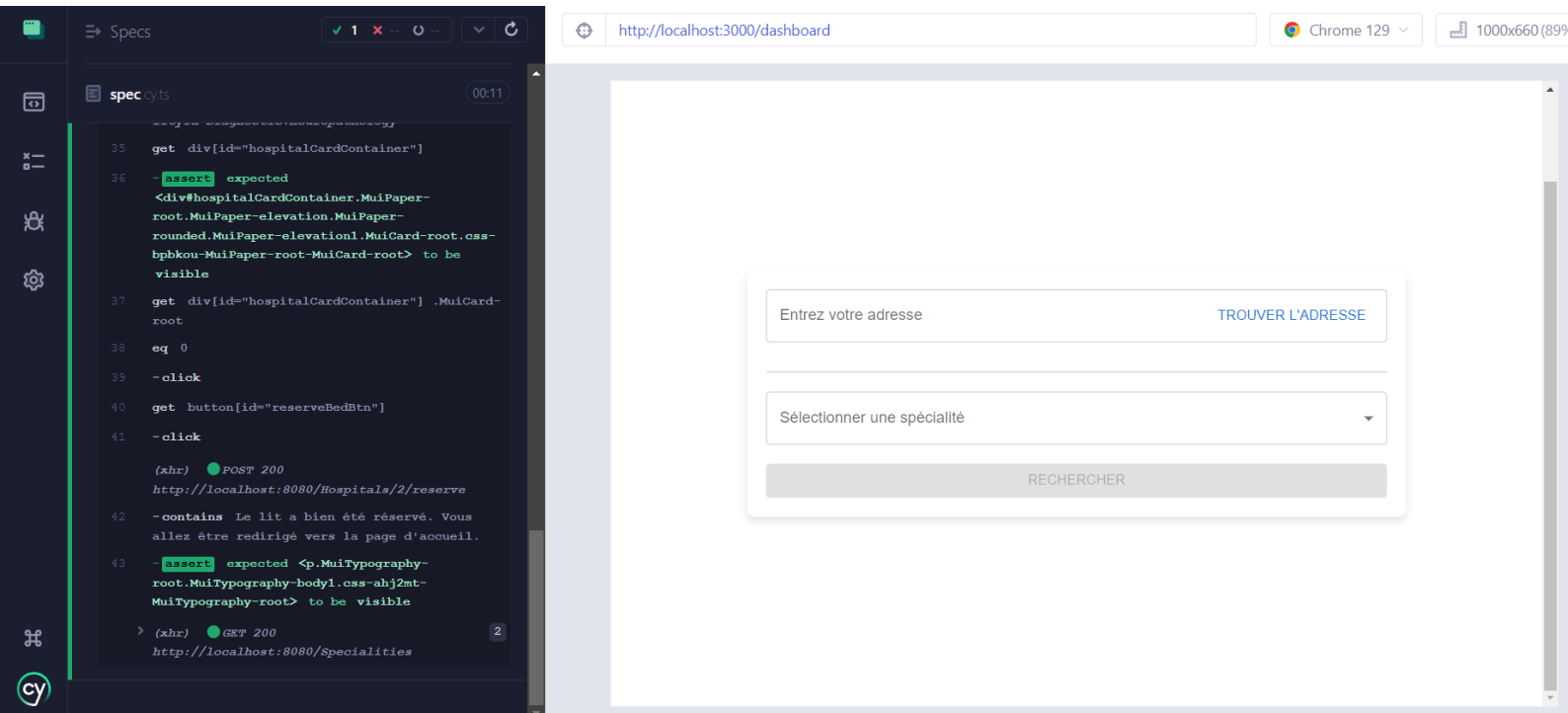
#### **Front-end**

Cypress est la technologie utilisée pour effectuer le test end-to-end du front-end.

Pour effectuer le test localement, il faut suivre la procédure suivante :

1. Se déplacer dans le dossier frontend et le lancer localement (le backend doit également être disponible).
2. Lancer la commande **npx cypress open**, et choisir le navigateur de son choix (ex : Chrome). Cela ouvre une nouvelle fenêtre.

### 3. Choisir le fichier de test à lancer. Le test E2E du frontend s'appelle **spec.ty.js**



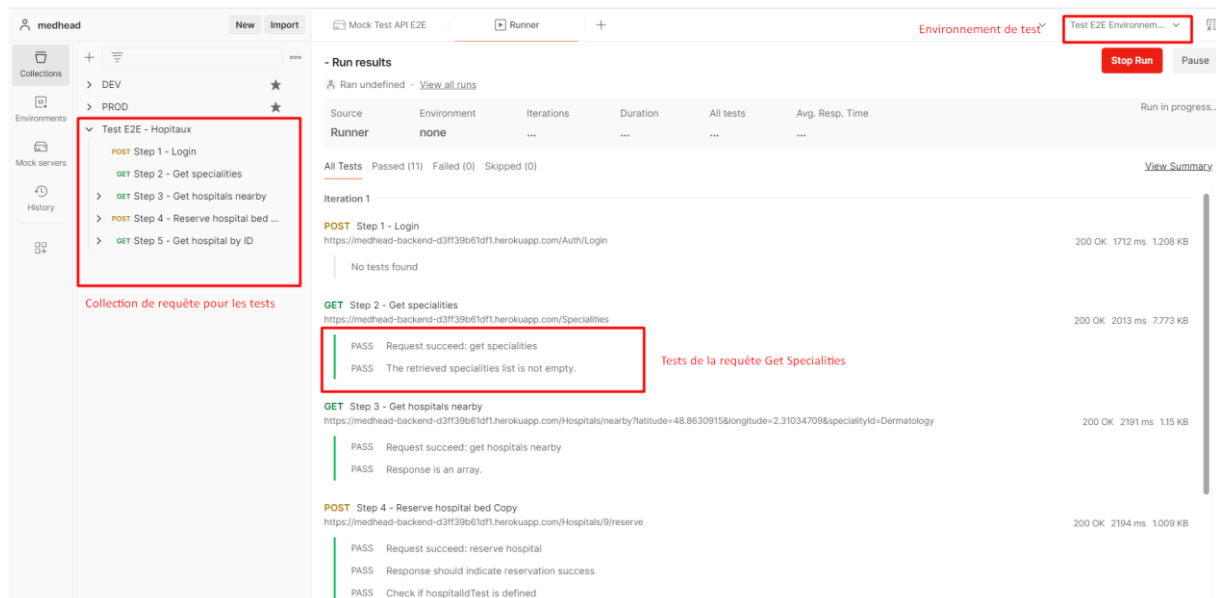
Fin d'exécution du test E2E du front-end

## Back-end

Pour effectuer un test end-to-end pour l'API backend, nous avons choisi **Postman**.

Postman permet de créer des collections de requêtes dans un environnement local et de les effectuer dans un ordre précis afin de simuler un chemin utilisateur.

Chaque requête peut posséder des tests associés (partie script).



Interface POSTMAN

Éléments importants :

- **Environnement** : Il faut sélectionner le bon environnement de test.

Test E2E Environnement

Save Fork 0 Share

Filter variables

	Variable	Type	Initial value	Current value	...
<input checked="" type="checkbox"/>	tokenTest	default	▼	eyJhbGciOiJIUzUxMiJ9.eyJ1c2VybWFnZSI6ImFuZHUyJldyI...	
<input checked="" type="checkbox"/>	specialtyIdTest	default	▼	Dermatology	
<input checked="" type="checkbox"/>	hospitalIdTest	default	▼	9	
	Add new variable				

### Environnement de test

- **Collection** : L'ensemble des requêtes. Le code source doit posséder un dossier postman avec les trois collections exportées ; **DEV, PROD, et Test E2E – Hopitaux**.
- **Tests de requête** : Chaque requête possède une section Script qui permet d'ajouter des tests.

GET https://medhead-backend-d3ff39b61df1.herokuapp.com/Specialities

Params Authorization Headers (7) Body Scripts Settings

Pre-request

Post-response \*

```
1 // Check if the request succeeded
2 pm.test("Request succeed: get specialities", function () {
3   pm.response.to.have.status(200);
4 });
5
6 // Verify that the array response is not empty
7 pm.test("The retrieved specialities list is not empty.", function () {
8   const jsonData = pm.response.json();
9   pm.expect(jsonData).to.be.an("array").that.is.not.empty;
10 });
11
12 // Search for a speciality with speciality_group equal to "Dermatology" and save its ID
   in env
13 const jsonData = pm.response.json();
14 const dermatologySpeciality = jsonData.find(speciality => speciality.speciality ===
   "Dermatology");
15
16 if (dermatologySpeciality) {
17   pm.environment.set("specialityIdTest", dermatologySpeciality.speciality);
18   console.log("Speciality id for Dermatology saved in environment: " +
   dermatologySpeciality.speciality);
19 } else {
20   console.log("No speciality found for Dermatology.");
21 }
```

### Exemple de script de la requête GET /Specialities

L'exécution de la requête permet de voir le résultat de celles-ci et voir si les tests ont été validés.  
Le test end to end du back-end effectue les requêtes suivantes :

1. Connexion
2. Récupération des spécialités
3. Récupération des hôpitaux proches

4. Réservation d'un lit d'hôpital
5. Récupération d'un hôpital

### Set de données

Le set de données des spécialités a été utilisé pour fournir la base de données.  
Pour les hôpitaux, seuls quelques exemples ont été créés.

## Sécurité

### Token JWT


Les token JWT permettent de protéger l'accès à l'API en mettant des restrictions sur les requêtes ;  
nécessité d'être un utilisateur connecté, rôle, ...

### HTTPS

Heroku propose une URL https par défaut pour les deux conteneurs.

### Cryptage

Les données sensibles (mot de passe des utilisateurs) sont encryptées en utilisant les méthodes  
BCryptPasswordEncoder lors de la création d'un utilisateur.

<input type="checkbox"/>	 id int8 ▾	username text ▾	password text ▾	+
<input type="checkbox"/>	3	testuser	\$2a\$10\$D1NbNawVhoja5nU5pcLf3uV7Gu	
<input type="checkbox"/>	5	andrew	\$2a\$10\$Z5oC6Ab2ZkwxhpzB3mMnL.C5i	

*Les mots de passes ne sont pas visibles directement depuis la base de données.*

## Résultats de la Proof of Concept

Le tableau suivant résume les exigences et justifie que la POC apportée les valide :

Exigence	Justification
La technologie Java est imposée par le consortium	Utilisation de java, springboot, maven pour le back-end.
L'API doit pouvoir s'inscrire à terme dans une architecture microservice.	Le code est structuré de façon à répondre à cette exigence. Il y a quatre microservices ; utilisateurs, hôpitaux, spécialités, authentification.
Interface graphique permettant de choisir la spécialité et de saisir la localisation	Le front-end présente une interface de login, un formulaire de choix de spécialité et d'adresse et une interface d'affichage des hôpitaux avec la possibilité de réserver un lit.
Utilisation d'une technologie moderne pour le front-end.	Utilisation de ReactJS et typescript.
Les données du patient sont protégées	Les mots de passe sont cryptés.
Les tests sont créés	Les tests unitaires, d'intégration, de charge et tests end to end sont effectués, documentés et les ressources accessibles.
La POC est facilement intégrable dans un développement future	Le code est disponible sur Github et versionné. Les ressources nécessaires pour déboguer et pour les tests sont également partagées ainsi que les workflows.
La POC est réutilisable	Le code est documenté.
Le code est versionné	Utilisation de Github.
La documentation technique de la PoC sera formalisée dans un fichier readme.md respectant le format markdown.	Il y a trois fichiers README rédigés en markdown ; un pour le dépôt central, un pour le back-end détaillant son fonctionnement et un pour le front-end.
Le document de reporting est fourni.	Document ci-présent.

## Points de surveillance

Cette section détaille les points à prendre en compte lors du développement de l'application pour la production :

- **Mise en place des environnements** : Il est important de mettre en place **au minimum** deux environnements (développement, production).
- **Automatisation du déploiement du back-end** : Actuellement, la CI/CD build et déploie une image docker automatiquement sur dockerhub mais il reste des commandes à faire à la main pour pousser ce répertoire dockerhub vers Heroku pour le back-end uniquement.

- **Compléter les tests** : Au fur et à mesure de l'ajout de fonctionnalités, il est important de créer les tests correspondants.
- **Documentation de l'API** : Il est nécessaire de documenter l'API, par exemple en utilisant un outil comme Swagger.
- **Plan de base de données** : La base de données utilisée pour la POC n'est pas complète et ne présente que peu de tables.
- **Plan de sauvegarde** : Il faut élaborer un plan de migration des données pour éviter les pertes lorsqu'il faudra effectuer la migration des données.
- **Disponibilité** : Utiliser un service cloud pour garantir la disponibilité de l'application.
- S'assurer que le **RGPD** soit toujours respecté

## Tester l'application

Les fichiers README des dossiers backend et frontend détaille les étapes pour lancer le projet en local, les prérequis nécessaires, et l'exécution des tests.

Ci-dessous un récapitulatif des commandes à faire pour exécuter les composants :

### Back-end

#### Prérequis :

- Java JDK 22
- Apache Maven
- IntelliJ ou autre IDE

#### Installation :

1. Clone le projet Devops-POC
2. Naviguer dans le bon dossier avec `cd backend`
3. Effectuer les commandes `maven compile` et `maven install`.
4. Lancer en local le fichier Jar généré dans le dossier `target/ backendMedhead-0.0.1-SNAPSHOT` en faisant un clic droit -> Run.

#### Debug :

Tester les différents endpoints en utilisant la collection DEV Postman fournie.

### Front-end

#### Prérequis :

- NodeJS (v20.15.1)
- Npm ou yarn
- git
- Un IDE comme VSCode

#### Installation :

1. Cloner le projet Devops-POC.
2. Naviguer dans le dossier du frontend avec `cd frontend`.
3. Installer les dépendances avec `npm install`.
4. Lancer le projet avec `npm start`.

### Debug :

Utiliser l'inspecteur Chrome / Firefox etc pour le debug d'éléments visuels.

Pour le debug de requêtes API, utiliser les collections POSTMAN fournies avant d'implémenter les requêtes dans le front-end afin de s'assurer du bon fonctionnement de celles-ci.

**Important : Sur l'interface du front-end, il y a un bouton Aide en dessous de la barre de navigation.**

**Ce bouton ouvre une popup avec les informations pour tester la fonctionnalité de recherche d'hôpitaux proches en donnant une adresse à chercher et la spécialité à sélectionner afin qu'il y ait au moins un hôpital retourné.**

**Pour entrer l'adresse, copier l'adresse donnée dans la popup et cliquer sur le bouton rechercher, puis sélectionner une des propositions.**

-> Le système a besoin que vous sélectionnez une adresse réelle proposée par l'API externe, car les coordonnées géographiques sont récupérées via l'API Openstreetmap.

## Liens utiles

Nom	Lien
Front-end	<a href="https://medhead-frontend-9f97491cebce.herokuapp.com/connexion">https://medhead-frontend-9f97491cebce.herokuapp.com/connexion</a>
Back-end	<a href="https://medhead-backend-d3ff39b61df1.herokuapp.com/">https://medhead-backend-d3ff39b61df1.herokuapp.com/</a>

## Conclusion

La preuve de concept a permis de répondre aux exigences précédemment définies.

L'API ainsi que le front-end sont hébergées et accessibles en ligne.

L'API est protégée par un système d'authentification, et les données sont protégées par un système de cryptage.

Le code a été écrit et documenté afin que d'autres équipes puissent le reprendre et le modifier le plus simplement possible. Les fichiers README permettent également à une personne nouvelle sur le projet de tester l'application, que ce soit pour le front-end ou le back-end.