MedHead

# Stratégie de tests

**- Proof Of Concept Medhead -**

**Par Wilfrid Boudia, Architecte logiciel**

**Le 20/03/2023**

# Table des matières

# A. Objet du document

Ce document explique les comportements attendus par le POC qui servira de base de travail pour le développement du nouvel outil de communication à mettre en place au sein de Medhead avec ses collaborateurs.

# B. Objectifs et tâches

## 1. Objectifs

L'objectif de ce document est de couvrir l'intégralité des tests et de les commenter. Nous pouvons nous appuyer sur les documents déjà fournis :

- Hypothèse de validation de principe
- Statement of Architecture Work
- Principes de l'architecture
- Solution Building Blocks

## 2. Méthodologie

Nous utilisons une approche de développement pilotée par les tests (TDD) dans laquelle nous définissons les résultats attendus d'une fonction répondant à un problème clair via un test qui vérifie que le résultat de la fonction corresponde à notre besoin. Puis nous élaborons la fonction qui répondra à ce besoin.

TDD garantit que le code source est soigneusement testé à l'unité et conduit à un code modulaire, flexible et extensible. Il se concentre uniquement sur l'écriture du code nécessaire pour passer les tests, rendant la conception simple et claire.

La couverture de code par les tests est assurée par Jacoco et mise dans des rapports à chaque build (/target/site/jacoco/jacoco-sessions.html). Nous espérons une couverture supérieure à 70%.

## 3. Tâches

Pour nous assurer que notre solution est entièrement validée par des tests automatisés, l'ensemble des tâches doit être opéré à chaque implémentation de fonctionnalité :

- Tests unitaires
- Tests fonctionnels
- Tests de performances et de stress
- Tests End To End / Test d'acceptation

A cela il faut ajouter des rapports facilement accessibles.

C'est pourquoi cette stratégie de test s'applique dans l'environnement de développement docker, avant (ou pendant) la compilation des images. Ce qui nous permet d'être sûr que toutes les machines lancées ont passé les tests. Cela s'applique de la même manière dans le pipeline CI/CD (avant de créer une nouvelle release ou de lancer un nouveau déploiement).

## 1. Description générales des tests

Les tests suivent une structure normée permettant d'offrir un maximum de possibilités au concepteur. Voici les principales étapes :

-@BeforeAll : sera exécuté en premier une seule fois (notamment pour calculer la durée des tests)

-@BeforeEach : sera exécuté avant chaque test (pour réinitialiser les données utilisées par exemples)

-@ParametizedTest : permettra de configurer un set de données à tester ou encore donner un nom plus explicite au test

-@AfterEach : sera exécuté à la fin de chaque test (pour réinitialiser une variable par exemple)

-@AfterAll : sera exécuté à la fin une seule fois.

## 2. Tests unitaires

**Définition :** En s'alignant sur la méthode Test-Driven Development (TDD), un test unitaire doit correspondre aux spécifications de l'application, il faut donc écrire les tests en premier puis les faire passer par la suite plutôt que d'écrire le code avant et de prendre le risque d'être influencé par celui-ci lors de la rédaction des tests.

**Couverture :** Toutes les fonctions autonomes, ne présentant pas de lien externe (fichier, bdd, requête, …) doivent être couvertes par les tests unitaires. Ils concernent également les controllers qui vont servir d 'échangeur de données avec le reste du système

**Méthodologie :** Les tests unitaires ont été conçus pour être les plus basiques possible afin de s'assurer que les fonctions les plus simples soient correctement exécutées.

Pour s'assurer de cela, ils ne sont utilisés qu'avec des données créées localement et lors du test pour s'assurer que la fonction en elle-même fonctionne correctement.

De plus, ils ne font appelle qu'à une seule fonction pour pouvoir retrouver cette dernière le plus rapidement possible.

## 3. Tests fonctionnels

**Définition :** Les tests fonctionnels vont vérifier la bonne compatibilité et les interconnexions entre les fonctions.

**Couverture :** Ces tests auront le même périmètre que les tests unitaires. Par contre, ils utiliseront des données externes afin de vérifier la capacité des fonctions à interagir avec les bases de données.

**Méthodologie :** Ces tests vont appeler des sets de données provenant d'une base extérieure afin de vérifier la bonne connexion avec les éléments externes à la fonction.

## 4. Tests End To End / Test d'acceptation

**Définition :** Les tests fonctionnels permettent, certes, de tester de nombreuses fonctionnalités d'une application, mais ils ne sont souvent pas suffisants pour tester complètement un projet. Les tests E2E vérifient l'intégralité de l'application, en jouant des scénarios prédéfinis, du début jusqu'à la fin. Ils permettent de voir si l'application répond correctement.

Dans notre cas, les tests End 2 End se transforment en tests d'acceptation automatisés.

Pour notamment des raisons de coût et de temps d'exécution de notre Pipeline CI/CD, nous devons limiter les scénarios présents dans les tests E2E. Pour cela, nous sommes limités à 2 tests pour limiter la charge de test.

**Scénario :** Nous envoyons les coordonnées géographiques (latitude, longitude) avec une spécialité afin de récupérer l'hôpital le plus proche et vérifier que l'envoi d'une demande de réservation à l'hôpital concerné à bien été pris en compte.

**Méthodologie :** Tout en restant dans son environnement de test, l'application E2E doit :

- Récupérer les paramètres
- Convertir le nom de la spécialité en id
- Charger les hôpitaux avec cette spécialité
- Sélectionné l'hôpital le plus proche
- Renvoyer le nom de l'hôpital
- Envoyer une demande de réservation de lit à l'hôpital concerné

### *5. Tests de performances et de stress*

**Définition :** Pour des tests de performances valides, il faut pouvoir vérifier que l'application pourra tenir la charge d'un grand nombre de requêtes à traiter pour être sûr qu'elles répondent qu'importent les circonstances.

**Scénario :** Nous allons effectuer un stress test de 10 secondes qui enverra 800 requêtes par seconde. Pour cela, nous utiliserons l'application JMeter configurée comme suit :

- un nombre de requêtes à 800 (Number of Threads)
- un nombre de période de montée en charge à 0 (Ramp-up period), pour un envoi immédiat
- une durée de 1 (Loop count), c'est-à-dire exécuté qu'une seule fois
- une minuterie constante à 1,25 milliseconde (Constant Timer), c'est-à-dire une requête envoyée toutes les 1,25 milliseconde

Les résultats sont consultables dans le dossier src/test/StressTest/ du projet.