PLAN DE TEST

Projet : Preuve de concept (l'allocation de lits d'hôpital pour les urgences)

Client : Consortium MedHead

Auteur : Loïc PIRIOU

MedHead

PLAN DE TEST

Table des matières

MedHead

PLAN DE TEST

1 Résumé du document

Ce document présente le plan de test mis en place pour le projet de la preuve de concept.

2 Version du document

Date	Version	Commentaires
29 mars 2023	0.01	Document de plan de test préliminaire.

3 Objectif du document

Ce document définit l'ensemble du plan de test qui sera mis en place pour le projet d'API de traitement des urgences.
Dans ce document nous présenterons la fonctionnalité, la méthodologie de test mise en place afin de garantir le succès de l'API.

4 Fonctionnalité

Le projet d'API de traitement des urgences vise la mise ne place de la fonctionnalités ci-dessous :

- Fournir une API RESTful qui tient les intervenants médicaux informés en temps réel sur l'hôpital le plus proche de l'intervention d'urgence en leur communiquant les éléments nécessaires pour la réservation d'un lit.

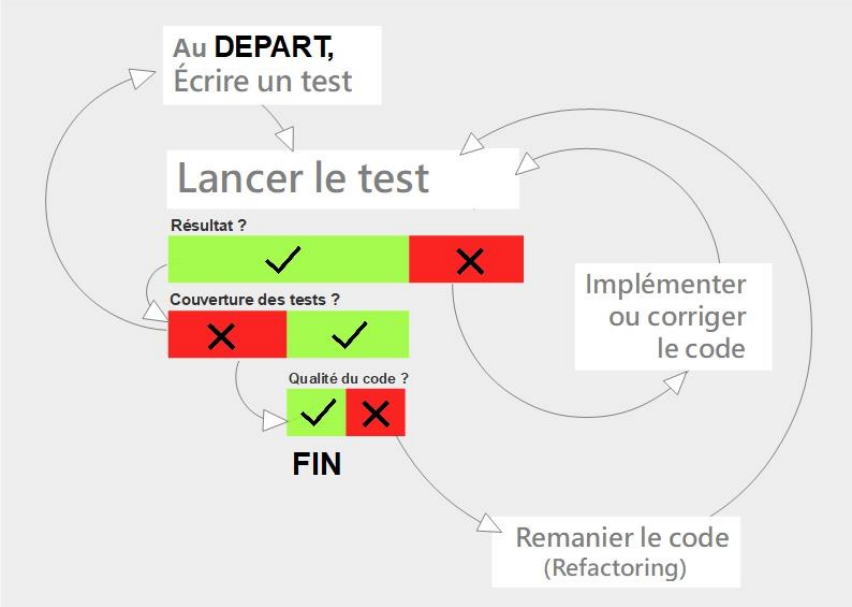
Ce projet doit permettre d'améliorer la prise en charge des patients afin de suggérer l'hôpital le plus proche offrant un lit disponible, associé à une ou plusieurs spécialisations correspondantes.

5 Méthodologie de Test

5.1 Test Driven Development (TDD)

Le Test Driven Development (TDD) place l'écriture des tests unitaires au rang de bonne pratique fondamentale dans la lutte contre ces risques. Le TDD préconise de commencer par l'écriture de tests, puis de procéder par itérations successives, alternant le lancement des tests avec l'implémentation ou la refonte du code livrable. La règle à suivre lors de la phase d'implémentation est de ne toujours implémenter que le minimum faisant passer le test. Une

phase de remaniement (refactoring) peut s'avérer nécessaire une fois que l'ensemble des tests passe au vert pour maximiser sa qualité.



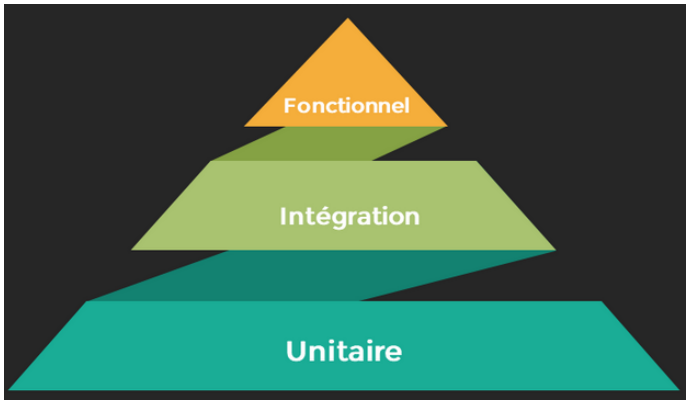
En s'appuyant uniquement sur les tests unitaires, par définition indépendants et reproductibles unitairement, la vérification de la bonne marche d'un comportement du logiciel qui serait la succession de comportements interdépendants dans un contexte précis n'est pas envisagée.

5.2 Behavior Driven Development (BDD)

Le Behavior Driven Design (BDD) étend le potentiel du TDD en proposant la définition de tests orientés comportement, écrits par un profil fonctionnel dans un langage naturel. Les cas de tests sont structurés selon le modèle Etant donné-Lorsque-Alors (nommé Gherkin). Chaque phrase est traduite sous forme d'une méthode, réalisant soit l'initialisation des conditions (Etant donné), soit les opérations (Lorsque), soit les assertions de comportements (Alors) définis par le scénario. Le test est déroulé dans l'ordre des phrases par appels successifs de l'ensemble de ces méthodes interdépendantes.
Elle incite à la collaboration des différentes parties prenantes au projet logicielle, équipes de développement, qualification et management en instaurant que le comportement d'une fonctionnalité sera décrit par des phrases basées sur un canevas composé de mots-clés du langage courant.
La communication est ainsi facilitée entre les équipes et évite des incompréhensions inhérentes à des parties d'environnements différents.
L'accent est mis sur les processus métiers auxquels le logiciel devra apporter des solutions.

5.3 Les types de Test

Les tests seront hiérarchisés via la pyramide de test ci-dessous :



Il y a donc trois types de tests à mettre en place :

- Les tests unitaires : ils testent que chaque fonctionnalité extraite de manière isolée se comporte comme attendu. De bons tests unitaires sont stables, c'est-à-dire que le code de ces tests n'a pas besoin d'être modifié, même si le code de l'application change pour des raisons purement techniques. Ils deviennent donc rentables, car ils ont été écrits une seule fois, mais exécutés de nombreuses fois. Les tests unitaires jouent un rôle clé dans l'amélioration de la qualité des logiciels, en détectant les erreurs plus rapidement, diminuant les erreurs en phase de maintenance et en améliorant la documentation du code source.
- Les tests d'intégrations : ils vérifient si les unités de code fonctionnent ensemble comme prévu. Ils en existent deux types :
    - Les tests d'intégration composants : ils permettent de vérifier si plusieurs unités de code fonctionnent bien ensemble, dans un environnement de test assez proche du test unitaire, c'est-à-dire de manière isolée, sans lien avec des composants extérieurs et ne permettant pas le démarrage d'une vraie application
    - Les tests d'intégration système : ils permettent de vérifier le fonctionnement de plusieurs unités de code au sein d'une configuration d'application, avec éventuellement des liens avec des composants extérieurs comme une base de données, des fichiers, ou des API en réseau.
- Les tests fonctionnels : ils visent à simuler le comportement d'un utilisateur final sur l'application, depuis l'interface utilisateur. L'ensemble du code développé est pris comme une boîte noire à tester, sans connaissance des briques et des unités de code qui la composent. Les simulations obtenues sont donc les plus proches de l'application utilisée dans des conditions réelles. Ces tests nécessitent toute l'infrastructure

nécessaire à l'application. Ces types de tests sont les plus lents à exécuter, et testent une partie beaucoup plus grande du code développé.

Des tests de charge seront également mis en place. Un test de charge consiste à effectuer un test permettant de mesurer la performance du système en fonction de la charge d'utilisateurs simultanées. L'objectif est de prévoir la charge maximale que peut encaisser l'API. Il permet également de mettre en évidence les points de vigilances du système, de les corriger et enfin de valider les performances de l'API.
Hormis la simulation sur le nombre de connexions simultanées, un test de charge permet également de tester le temps de réponse du système, sa robustesse, ou encore de dimensionner des serveurs etc …
En d'autres termes, il existe deux types de tests de charge :

- Le test de performance: qui permet de mettre en évidence les points sensibles et critiques de l'architecture technique. Les métriques qui sont prises en compte dans ce type de test de charges sont les suivantes : temps de réponse, charge système, requête de base de données, etc.
- Le test aux limites: qui permet de déterminer et de prévoir la capacité maximale que peut encaisser un SI, lorsqu'une application est testée avec une activité nettement supérieure à une activité normale.

6 Tests des composants

6.1 Microservice Authorization

Test n°	Détails du test	Composant(s) mobilisé(s)
1	Etant donné un appel sur l'API de connexion Lorsque l'utilisateur est reconnu Alors un jeton JWT est renvoyé	Microservice Authorization BDD Utilisateur
2	Etant donné un appel sur l'API de connexion Lorsque l'utilisateur n'est pas reconnu Alors un json d'erreur est renvoyé	Microservice Authorization BDD Utilisateur
3	Etant donné un appel sur l'API de vérification d'un JWT Lorsque celui-ci est reconnu Alors un booléen True est renvoyé	Microservice Authorization
4	Etant donné un appel sur l'API de vérification d'un JWT Lorsque celui-ci n'est pas reconnu Alors un booléen False est renvoyé	Microservice Authorization

6.2 Microservice Emergency

6.2.1 Controller

Test n°	Détails du test	Composant(s) mobilisé(s)
1	Etant donné un appel sur l'API d'urgence Lorsque l'utilisateur a communiqué un jeton d'authentification valide, et un objet Emergency valide Alors un objet NearestHospitalReservation est renvoyé	Ms Emergency Controller Ms Emergency Service Ms Emergency Repository
2	Etant donné un appel sur l'API d'urgence Lorsque l'utilisateur a communiqué un jeton d'authentification valide, et un objet Emergency non valide Alors l'API renvoi une erreur	Ms Emergency Controller
3	Etant donné un appel sur l'API d'urgence Lorsque l'utilisateur a communiqué un jeton d'authentification non valide Alors l'API renvoi que l'utilisateur est non autorisée	Ms Emergency Controller Ms Emergency Repository

6.2.2 Service

Test n°	Détails du test	Composant(s) mobilisé(s)
1	Etant donné un appel sur la fonction getNearestHospitalReservation Lorsque celle-ci reçoit un objet Emergency contenant la spécialité et les coordonnées d'urgence Alors elle renvoie un objet NearestHospitalReservation contenant le nom de l'hôpital, ses coordonnées et le numéro de réservation du lit	Ms Emergency Service Ms Emergency DTO Ms Emergency Repository
2	Etant donné un appel sur la fonction getNearestHospital Lorsque celle-ci a un objet Emergency et une liste d'hôpitaux Alors elle renvoie l'hôpital le plus proche de l'urgence	Ms Emergency Service Ms Emergency Repository

6.2.3 Repository

Test n°	Détails du test	Composant(s) mobilisé(s)
1	Etant donné un jeton d'authentification communiqué dans l'appel à l'API d'urgence Lorsque la fonction isTokenAvailable est appelé pour transmettre le jeton au MS Authorization Alors elle renvoi un booléen sur la validité du token	Ms Emergency Repository
2	Etant donné un appel sur la fonction savedBedAvailable avec l'Id d'un hôpital Lorsque celle-ci transmet l'Id de l'hôpital au MS BedAvailable Alors celui-ci renvoi un numéro de réservation pour un lit disponible dans l'hôpital lié à l'Id	Ms Emergency Repository
3	Etant donné un appel sur la fonction getAvailableHospitalsBySpecialist avec une spécialité Lorsque celle-ci transmet la spécialité au MS Hospital Alors celui-ci renvoi une liste d'hôpital ayant des lits disponibles pour la spécialité demandé	Ms Emergency Repository
4	Etant donné un appel sur la fonction getDistanceBetweenHospitalAndEmergency avec les coordonnées d'un hôpital et les coordonnées de l'urgence Lorsque celle-ci transmet se coordonnées à l'application Graphhopper Alors celui renvoi un json avec plusieurs informations dont la distance entre les deux coordonnées	Ms Emergency Repository

7 Tests fonctionnels

L'ensemble des fonctionnalités est rappelé dans chacun des tableaux ci-dessous avec l'ensemble des tests fonctionnels liés.

Fonctionnalité n°1	Tenir les intervenants médicaux informés en temps réel sur l'hôpital le plus proche de l'intervention d'urgence en leur communiquant les éléments nécessaires pour la réservation d'un lit.	Composant(s) mobilisé(s) pour le test
Test Fonctionnel 1	Etant donné un appel sur l'API d'urgence Lorsque l'utilisateur a communiqué un jeton d'authentification valide, et un json valide contenant la spécialité d'urgence et les coordonnées Alors l'hôpital le plus proche de l'urgence ayant un lit de disponible avec la spécialité est renvoyé avec un numéro de réservation pour le lit	Microservice Emergency Microservice Authorization Microservice Hospital Microservice BedAvailable Application Graphhopper
Test Fonctionnel 2	Etant donné un appel sur l'API d'urgence Lorsque l'utilisateur a communiqué un jeton d'authentification valide, et un json non valide Alors le microservice renvoi une erreur	Microservice Emergency Microservice Authorization
Test Fonctionnel 3	Etant donné un appel sur l'API d'urgence Lorsque l'utilisateur a communiqué un jeton d'authentification non valide Alors un retour non autorisée est envoyé	Microservice Emergency Microservice Authorization

Fonctionnalité n°2	Authentification d'un utilisateur	Composant(s) mobilisé(s) pour le test
Test Fonctionnel 1	Etant donné un utilisateur transmettant son identifiant et son mot de passe Lorsque l'identifiant et le mot de passe sont reconnues par le MS Authorization Alors l'utilisateur est connecté et le MS Authorization renvoi un jeton d'authentification	MS Authorization
Test Fonctionnel 2	Etant donné un utilisateur transmettant son identifiant et son mot de passe Lorsque l'identifiant et le mot de passe ne sont pas reconnues par le MS Authorization Alors l'utilisateur n'est pas connecté et le MS Authorization renvoi l'information que l'utilisateur n'est pas autorisé à se connecter	MS Authorization

8 Collecte des données des tests

Les tests unitaires sont des tests permettant de tester de toutes petites unités d'un logiciel. Ils sont notamment dédiés à des tests de méthodes.
Les tests unitaires doivent être lancés aussi souvent que possibles, notamment à chaque modification du code source: il faut donc qu'ils soient exécutables automatiquement. JUnit est une famille de bibliothèque de tests unitaires. Junit 5 est pris en charge par les outils de build (Maven). Pour les tests unitaires Web, nous pouvons utiliser la brique SpringBoot de test.
Cette même brique (SpringBoot Test) peut être utiliser afin de créer et de générer nos tests d'intégrations.
Pour l'édition des rapports des tests unitaires et d'intégrations nous utiliserons Maven et les plugins Surefire et Jacoco, celui-ci donnant le taux de couverture des tests.
L'ensemble de ces tests peuvent être lancés automatiquement via un pipeline GitHub Actions sur le repository du code. On liera le repository à un compte SonarQube pour vérifier la qualité du code, connaître le taux de couverture et savoir si le code est déployable en production. Cette pipeline peut être configuré pour se lancer à chaque commit, pull request. Si l'ensemble de la pipeline s'exécute correctement alors un artifact du repositiory sera édité.
Pour effectuer les tests de charge nous utiliserons le logiciel Jmeter qui est un outil efficace pour les tests de performances. Il peut être utilisé pour simuler le trafic des utilisateurs, trouver des problèmes et suivre les résultats. Il peut être utilisé pour simuler une charge importante sur un serveur, un ensemble de serveurs, un réseau ou d'autres objets afin de tester la charge de l'API et d'évaluer les performances globales de celle-ci.