Document de Stratégie de test

Frédéric Pichot 27/10/2021

Table des Matières

Table des figures

1. Information sur le document

Ce document a pour vocation de lister les hypothèses de développement d'une preuve de concept pour le sous système d'intervention d'urgence.

Nom du projet	Preuve du concept
Préparé par :	Frédéric Pichot
N° de version du document :	0.1
Titre :	Document de stratégie de test
Date de version du document :	07/10/2021

Figure 1: Information sur le document

2. Liste de distribution

De	Date	Phone/Fax/Email
Ash Kara	27/10/2021	Email
Chris Pike	27/10/2021	Chat

Figure 2 : Liste de distribution

3. Historique des versions du document

Numéro de Version	Date de la version	Revu par	Description	Nom du fichier
1	27/10/2021	Frédéric Pichot	Création du document	Document de stratégie de test

Figure 3 : Historique des versions du document

4. Environnement de développement

Outils & système	Dénomination
Sytème d'exploitation	Ios
Environnement de développement	Eclipse
Langage de programmation	Java
Machine virtuel Java	OpenJDK 11
Framework	Spring boot
Plugins Eclipse	Spring Tools 4, Maven, Git
Test unitaire	JUnit, Mockito
Test fonctionnel	Gherking, Serenity, Selenium
Conteneur	Docker
Système de versioning	GitHub

Figure 4 : Environnement de développement

5. Stratégie de tests

5.1 Les tests unitaires

5.1.1 JUnit

Les tests unitaires JUnit appliqueront le principe FIRST dont voici la définition de l'acronyme.

F (Fast) : Fast en anglais signifie rapide. Les tests unitaires servent à surveiller l'état de fonctionnement des différents composants de l'application. Ils seront très nombreux et par leur nombre leur temps de fonctionnement doit être très court.

I (Isoled) : L'anglicisme explicite décrit l'indépendance de chacun des tests aux regards des autres. Leur point de contrôle doit-être le plus petit possible et ne doit pas être un vassal d'un autre test.

Cette indépendance permet de localiser très précisément l'endroit du problème. Pour faciliter la recherche et donc le temps de panne, les développeurs devront apporter un soin particulier aux dénomination de leur test.

Un nom de méthode doit avoir son équivalent dans la zone de tests qui s'autorisera d'être plus explicite dans le cas ou la partie à évaluer serait trop volumineuse. En effet, il est fortement conseillé de la découper en plusieurs petites actions de test. Les tests doivent aussi contrôler la phase négative de la méthode. (On peut avoir des tests dont le résultat est correcte lorsque la valeur attendue est une erreur).

Ils ne devront pas dépendre d'élément externe ni être conditionnés par quoique ce soit.

Les commentaires devront faciliter la compréhension de la démarche du test, de plus les messages d'erreurs devront-être explicites et indicatif du lieu du dysfonctionnement.

R (Repeatable): Ils devront-être ré-exécutable avec un rythme qui reste à déterminer. De plus leur cycle de fonctionnement ne doit par perturber l'outil.

S (Self-validation): Les tests doivent-être en capacité de s'auto-validation, c'est-à-dire que le test ne laisser aucun doute sur le retour de leur action tant dans le domaine du succès que dans celui de l'échec.

T (Timely): Le test doit-être rédigé juste avant la phase qu'il est en charge d'évaluer. Ce principe s'inscrit dans l'approche Test-Driven-Developpement(TDD), cela sous-entend que lorsque le test est terminé, il doit est en erreur. Puis le développeur doit coder de façon à le rendre valide.

Un fois accompli, ce sera le bon moment pour re-factoriser son code.

5.1.2 Mockito

Mockito est un framework de test, il permet la création d'objets doubles de test (objet fictif) dans des tests unitaires à des fin de permettre le développement piloté par les tests (TDD). En effet il simule le comportement pour que le cycle du test soit possible.

Il permet aux développeurs de vérifier le comportement du système sous test sans établir d'attente au préalable.

5.2 Test d'intégration des composants

Le test d'intégration a pour cible de détecter les erreurs non détectables par les tests unitaires. Il permet également de vérifier l'aspect fonctionnel, les performances et la fiabilité du logiciel.

L'intégration fait appel en général à un système de gestion de versions et éventuellement à des programmes d'installation. Cela permet de contrôler la fonctionnalité d'une nouvelle version avant qu'elle soit lancé en production.

6. Liste des tests à inclure

6.1 Les fonctionnalités

6.1.1 Interrogation des disponibilités des hôpitaux en lien avec la pathologie du patient

EN TANT QUE => Utilisateur

JE VEUX => Trouver un lit disponible

POUR CE FAIRE => Lorsque je saisie la pathologie

ALORS => Une liste d'hôpitaux doit apparaitre

PHASE DE CONTRÔLE => au retour ils sont classées par nombre de lits libres

6.1.2 Classement croissant de celles-ci en fonction de la distance

EN TANT QUE => Utilisateur

JE VEUX => Trouver l'hôpital le plus proche

POUR CE FAIRE => Lorsque j'entre mes coordonnées

ALORS => Une liste d'hôpitaux doit apparaitre

PHASE DE CONTRÔLE => au retour ils sont classées en lien avec la distance

6.1.3 Si pas de disponibilité déroger aux règles de distance selon la gravité

EN TANT QUE => Utilisateur

JE VEUX => Pouvoir déroger aux règles de distance

POUR CE FAIRE => Si pas de lit disponible au plus près

ALORS => Je clique sur « élargir la recherche »

PHASE DE CONTRÔLE => une liste d'hôpitaux apparait classée par distance

6.1.4 Si pas de disponibilité opter pour l'hôpital le plus proche qui dispose de lits vacants

EN TANT QUE => Utilisateur

JE VEUX => Obtenir un lit disponible

POUR CE FAIRE => Lorsqu'il n'y a pas de place pour la pathologie

ALORS => Je clique « rechercher lits disponibles »

PHASE DE CONTRÔLE => une liste d'hôpitaux apparaît classée par distance

6.1.5 Pré réservation de l'hôpital et mise à jour du nombre de lit disponible.

EN TANT QUE => Utilisateur

JE VEUX => Réserver le lit disponible

POUR CE FAIRE => Lorsque je clique sur l'hôpital choisie

ALORS => Un ordre de réservation est envoyé

PHASE DE CONTRÔLE => Le nombre de lit disponible est réactualisé

6.1.6 Tri selon les symptômes pour intervention d'urgence

EN TANT QUE => Utilisateur

JE VEUX => Pouvoir ordonnancer les priorités

POUR CE FAIRE => Lorsque plusieurs demandes arrivent

ALORS => Je clique sur « classer par priorité »

PHASE DE CONTRÔLE => Les demandes classées par priorités

6.2 Listes des contrôles

Pour éviter que l'application souffre d'une dette technique, l'architecture sera organisée autour d'un ensemble de micros service. Pour quelle soit conforme effectuer les contrôles suivants :

- Vérifier que l'équipe de développement dispose de toutes les compétences nécessaires, autrement élaboré un plan de formation dédié
- Catégorisé et organisé les modules à développer
- Chaque fonction doit-être scrupuleusement définie et documentée
- Développer selon les préceptes du TDD.
- Le plan de test est complet et bien suivi
- Les scénarios sont exécutés et produisent des scénarios d'acceptation
- Vérifié que chaque micro-service ont été testé au travers d'outil comme Postman.
- Corrélé que les fonctionnalités de l'application répondent aux exigences et au attendus
- Edité et ratifié les conventions SLA

6.3 Liste des composants et des tests

Liste des composants

- Gestion des dossiers médicaux
- Gestion des spécialités
- Gestion des spécialistes médicaux par spécialité
- Gestion des prestataires médicaux
- Gestion des rendez-vous
- Gestion des disponibilités par spécialité
- Gestion de la cartographie des possibilité

Typologie des tests

6.3.1 Test de l'interface utilisateur

Les tests se concentreront sur les retours de l'API, ils doivent révéler l'efficience du front-end et du back-end. Ils devront présenter une vue d'ensemble des moyens mis à disposition aux organismes santé. Cela n'empêche pas une certaine présentation et une convivialité à l'utilisation.

6.3.2 Test de charge

Les tests de charge se feront lorsque l'outils sera terminé et porté dans la phase de production. Ils s'effectueront dans des contions de normalité, mais aussi et surtout dans des conditions de forte sollicitation. Ils devront révéler la pertinence des choix théoriques.

6.3.3 Test de sécurité

Les tests de sécurité doivent certifier que l'API est protégée contre les menaces. Ils doivent valider le mode de cryptage et la gestion des droits des utilisateurs en fonction de leur niveau d'accréditation.

6.3.4 Test de pénétration

Ces tests permettront d'évaluer le niveau d'accès aux ressources, aux processus et d'estimer le degré de résistance à leur corruption.

6.3.5 Test de Fuzz

Fuzz est un moyen de rechercher les limites du système jusqu'au crash de l'outil. Pour y parvenir on injecte une très grande quantité de données jusqu'à le saturer. Cette information permettra d'élaborer des stratégies d'anticipation et de gestion de crise.

6.3.6 Evaluations du temps d'exécution et des pertes

Avec l'insertion de log, il sera possible d'estimer les temps de latence entre la demande et les retours de l'API, ces information seront à mettre en perspective avec la fluidité du réseau. Nous obtiendront aussi un vu quantitative et qualitative des erreurs et ainsi visualiser le volume de fuites des ressources.

6.4 Rapport d'évaluation

Chaque test fonctionnel génèrera un rapport qui permettra de visualiser sa validité. En cas d'échec les erreurs générées seront dotées d'un moyen de les localiser. Les rapports seront édités au format HTML, stockés et ventilés par fonctionnalités.