Stratégie de test

Comme déclaré dans le document “Hypothèse de validation de principe”, le sous-système d'intervention d'urgence en temps réel, ERS (Emergency Responder System) est destiné à recevoir une ou plusieurs spécialités médicales et une banque de données d'informations récentes sur les hôpitaux afin de suggérer l'hôpital le plus proche offrant un lit disponible, associé à une ou plusieurs spécialisations correspondantes. Le lieu de l'incident d'urgence doit également être fourni. Dans ce même document nous avons établi des exigences et des kpi extrêmement élevés, c’est pourquoi notre stratégie de test doit être complète et fiable.

Project Name:	ERS, Emergency Responder System (allocation de lits d'hôpital pour les urgences)		
Prepared By:	Damien Senechal	**Document Version No:**	0.1
Title:	Stratégie de test	**Document Version Date:**	
Reviewed By:		**Review Date:**	15/06/22

Table des matières

Objectifs et tâches

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

Méthodologie

- Nous utilisons une approche de développement pilotée par les tests (TDD) dans laquelle un test est écrit avant d'écrire le code. Une fois que le nouveau code réussit le test, il est refactorisé en une norme acceptable.
- TDD garantit que le code source est soigneusement testé à l'unité et conduit à un code modulaire, flexible et extensible. Il se concentre sur l'écriture uniquement du code nécessaire pour passer les tests, rendant la conception simple et claire.
- Nous utilisons au maximum les notations pour écrire les tests, celles issue du BDD (Behaviour-driven development) sous forme de : Given (étant donné)/When (quand)/Then (alors).
- La couverture de code par les tests est comptée et mise dans des rapports à chaque build (CF. Jacoco). Nous espérons une couverture supérieure à 80%.

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 d'intégration système
- 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 la pipeline CI/CD (avant de créer une nouvelle release ou de lancer un nouveau déploiement).

Stratégie de test

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.

Méthodologie :

- Utilisation des notations sous forme : Given / When / Then.
- Prise en compte des cas limites (« edge cases ») qui sont des tests conçus pour vérifier l'inattendu aux limites de votre système et de vos limites de données et les cas pathologiques (« corner cases ») qui ont lieu lorsque plusieurs cas limites se présentent ou se mélangent avec des dysfonctionnements extérieurs.
- Le risque zéro n'existe pas. Réduisez le risque en combinant les scénarios attendus, les cas limites courants, et les risques de non-disponibilité des services extérieurs dont vous dépendez ;
- Le test va au-delà d'une simple manière de corriger le code, c'est un véritable état d'esprit agile.

Tests d'intégration système

Définition : Les tests d'intégration système 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.

Couverture : Puisqu'il s'agit principalement d'une API RESTFul, toutes les fonctions contenants des requêtes entrantes ou sortantes doivent être traitées dans leurs environnement (au plus proche de celui de production) avec la possibilité de prendre des informations extérieures. Dans le document Solution Building Blocks prochainement écrit, nous allons suggérer d'établir un projet qui aurait pour rôle de simuler l'API Hospital pour ne pas tester sur des API externes avec des données réelles (confidentialité NHS). Par contre l'API de calcul de distance et de temps de route entre deux points, peut être utilisée durant les tests.

Méthodologie : Idem

Tests End To End / Test d'acceptation

Définition : Les tests d'intégration 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 raison de coût et de temps d'exécution de notre Pipeline CI/CD, nous devons limiter les scénarios présents dans les tests E2E. 2 ou 3 scénarios, avec 4 ou 5 variantes seraient un maximum (à déterminer selon le budget et le temps d'exécution).

Scénario :

- Requête de demande de réservation de lit dans un hôpital le plus proche
    - variante classique : utilisateur lambda, informations corrects et communication avec l'API Hospital fonctionnelle
    - variante perte 3rd party : idem que précédemment à l'exception que l'on coupe la communication avec l'API Hospital

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

1. Exécuter la requête BedAvailability sur le point d'entré classique soit la gateway
2. Joindre le token d'authentification, ou pas
3. Joindre les informations (de localisation, de spécialités NHS), ou pas
4. La requête doit atterrir sur l'API Rest
5. La communication avec l'API Hospital (externe) fonctionne ou pas
6. Retourner une réponse cohérente quoiqu'il advienne

Tests de performances et de stress

Définition : Pour des tests de performances, il faut pouvoir tout tester en même temps et tracer tout ce qui s'y passe dans des fichiers de logs. Pour cela nous nous en remettons à notre plateforme cloud Amazon Web Service.

Le test de charge distribué sur AWS (DLT) nous aide à automatiser les tests de performances de nos applications à grande échelle. De cette façon, nous pouvons identifier les goulots d'étranglement avant de publier notre application. La solution prend en charge les tests planifiés et simultanés et gère un nombre accru d'utilisateurs simultanés qui peuvent générer de nombreuses requêtes par seconde. Il offre également des rapports en temps réel. Avec DLT, nous pouvons simuler des milliers d'utilisateurs se connectant à notre application, afin de mieux comprendre le profil de performances de notre application. Il utilise des scripts JMeter pour créer des tests personnalisés et Taurus, un framework de test de charge open source pour automatiser les tests.