2022

Stratégie de Test
POC Emergency Responder Subsystem

Magalie Morteau
Architecte Logiciel
MedHead
20/01/2022

Table of contents

1 Objectif du document 2

2 Contexte 3

3 Approche 4

3.1 Approche générale 4

3.2 Approche détaillée 5

4 Stratégie de test 7

4.1 Pyramide des tests 7

4.2 Tests unitaires 8

4.3 Tests d'intégration 9

4.4 Tests E2E 9

4.5 Tests de performance 10

4.5.1 Tests de performance 10

4.5.2 Tests de charge 10

4.5.3 Tests de stress 10

4.6 Tests de sécurité 11

4.7 Test de qualité 12

4.8 Critères d'échelonnement 12

5 Outils de test 13

1 Objectif du document

Ce document décrit l'approche et la stratégie de test qui seront mis en place dans le cadre des tests du projet.

La stratégie de test permet la validation des principes référencés dans le document « Principes de l'architecture ».

La stratégie de test est la description de haut niveau des exigences de test à partir de laquelle un plan de test détaillé sera ensuite dérivé, spécifiant des scénarios de test individuels et des cas de test.

2 Contexte

Emergency Responder Subsystem (ERS) est un système optimisé qui permettra de gérer les demandes d'intervention urgentes depuis la future plateforme des différents intervenants de MedHaed.

Ce sont des systèmes critiques pour la sécurité des patients.

Les parties prenantes ont des craintes quant à la gestion des services d'urgence ; elles veulent être sûres à 100 % que le service fonctionnera correctement, même pendant les périodes de pic d'activité, qu'il sera en capacité de fournir une allocation de lits en temps opportun.

Les préoccupations des principales parties prenantes doivent être prises en compte et être prioritaires, en particulier en ce qui concerne les temps de réponse, l'évolutivité, la tolérance aux pannes des systèmes hospitaliers auxiliaires et la résilience sous charge.

Nous pensons que la mise en œuvre d'une preuve de concept pour le sous-système d'intervention d'urgence en temps réel par l'équipe d'architecture métier du Consortium MedHead permettra :

- d'améliorer la qualité des traitements d'urgence et de sauver plus de vies ;
- de gagner la confiance des utilisateurs quant à la simplicité d'un tel système.

Il s'agit d'un POC permettant de vérifier que le sous-système ERS de la plateforme médical de MedHead sera hautement optimisée et que ce système aura des niveaux élevés de tolérance aux pannes.

Ce POC va permettre d'atténuer les risques liés à la solution proposée, avec des actions en prévention mis en place dans ce POC, ce qui permettra de rassurer les parties prenantes du consortium.

3 Approche

3.1 Approche générale

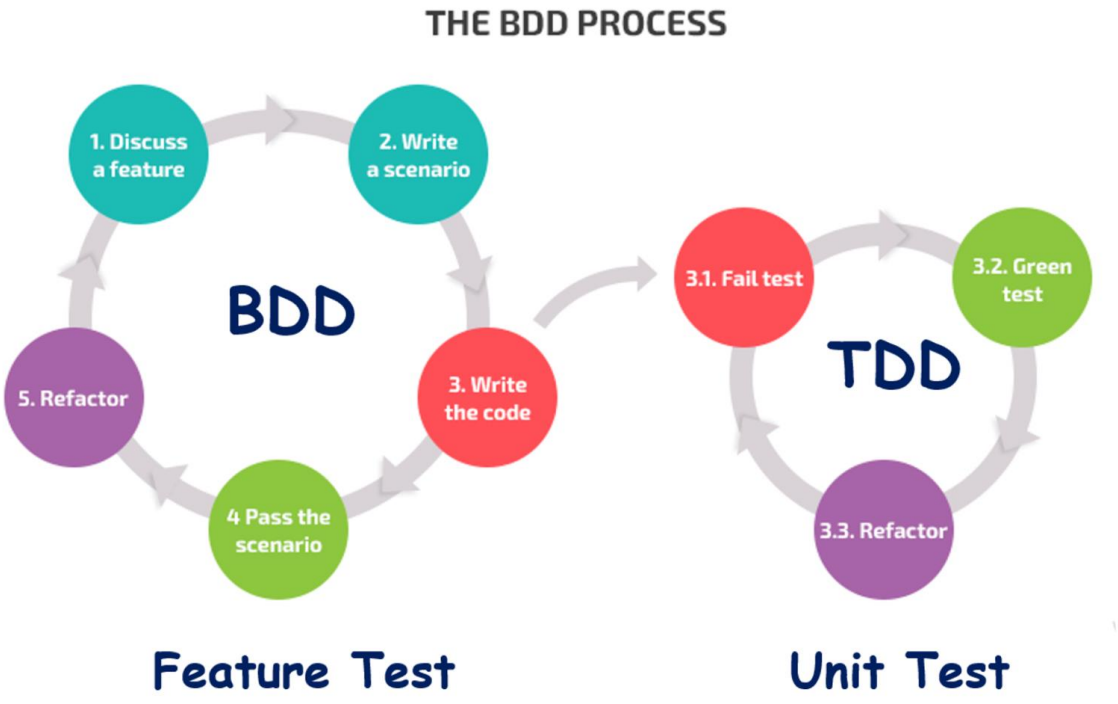
Nous appliquerons un développement centré sur le comportement selon la méthodologie Business-Driven-Development (BDD).

- Pour garantir l'exactitude des résultats attendus centrés sur le patient.
- Pour soutenir un développement aligné avec un langage omniprésent.

Dans l'approche de test en BDD ce sont les règles métiers qui structurent les tests.

Le processus des tests en BDD :

- Reconnaître une caractéristique métier.
- Catégoriser les scénarios sous la fonctionnalité sélectionnée.
- Définir les étapes pour chaque scénario.
- Exécuter la fonctionnalité, elle est en échec.
- Écrire du code pour faire passer le test.
- Refactoriser le code et créer une bibliothèque d'automatisation réutilisable.
- Exécuter la fonctionnalité pour qu'elle passe.
- Générer les résultats des tests.



Dans le processus en BDD on met en place des tests automatisés précoces, complets et appropriés.

Ce principe encourage l'utilisation de techniques de développement dirigé par des tests (TDD pour Test-Driven Development). Afin de valider rapidement les exigences, il est recommandé d'utiliser le langage du domaine métier lors des tests.

3.2 Approche détaillée

La stratégie de test pour le POC ERS consiste à établir un plan de test qui devra respecter les principes d'architecture résumés dans ce chapitre.

Continuité des activités des systèmes critiques pour les patients (cf. Principe B1)

- Tests incrémentiels de performances, de vulnérabilité et d'exposition pour chaque incrément de la plateforme technique.
- Tester les plans de reprise qui devront exister pour tous les systèmes critiques.

Clarté grâce à une séparation fine des préoccupations (cf. Principe B2)

Le découpage en micro service permettant de tester individuellement chaque service de l'architecture.

Intégration et livraison continues (cg. Principe B3)

Les exécutions CI/CD génèrent des journaux ou des sorties clairs qui peuvent être analysés pour isoler les builds en échec ou les erreurs dans les étapes de build, de test et de livraison.

Tests automatisés précoces, complets et appropriés (cf. Principe B4)

Les applications doivent être construites à l'aide de tests automatisés qui garantissent la fiabilité à la fois fonctionnelle et non fonctionnelle de la mise en œuvre.

Les bogues logiciels sont inévitables et peuvent être causés par des erreurs de code ou d'analyse. Des tests précoces garantissent que le logiciel est construit selon les spécifications et que chaque spécification est validée avant d'investir dans de mauvaises solutions.

Ce principe encourage l'utilisation de techniques de développement dirigé par des tests (TDD pour Test-Driven Development).

Les équipes doivent suivre la pyramide des tests et mettre en œuvre un niveau de test approprié pour chacune des catégories de tests suivantes

- Unitaire
- Intégration
- E2E (End To End : de bout en bout)

Lorsque les services sont interdépendants, il est également conseillé d'envisager des tests centrés sur le consommateur.

Sécurité de type « shift-left » (cf. Principe B5)

Tests de sécurité continus et automatisés pour réduire le risque dû à une erreur ou à une omission humaine.

Favoriser une culture de "learning" avec des preuves de concept, des prototypes et des Spike (cf. Principe C4)

- Isoler les preuve de concept des données et des systèmes de production

  L'apprentissage peut être mené de manière isolée dans un environnement artificiel afin d'éviter l'impact sur les systèmes de production. Utiliser des données factices ou anonymisées

- Les tests de performance des prototypes et des implémentations centrées sur l'apprentissage devraient valider les algorithmes clés faisant partie de cette échelle d'apprentissage.

- Plan de test comme outils de communication des exigences

  Un système lisible à la fois pour les humains et les ordinateurs est utilisé. Pour rédiger les cas de test BDD pour un Use Case on utilisera la syntaxe du modèle Gherkin avec la formule Given-When-Then.

  The **Given** scenario + **When** an action takes place + **Then** this should be the outcome.

- Tester les rapports d'exécution pour documenter le comportement pris en charge:

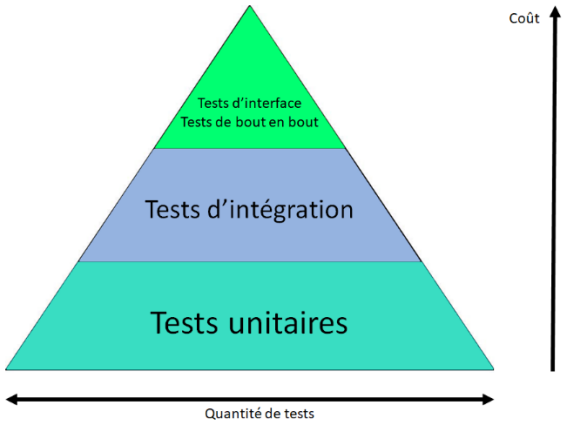
  Les PoC doivent avoir des pipelines CI qui exécutent des tests et produisent des rapports d'exécution des tests.

4 Stratégie de test

4.1 Pyramide des tests

S'assurer que le PoC est entièrement validé avec des tests d'automatisation reflétant la pyramide de test (tests unitaires, d'intégration, d'acceptation et E2E) et avec des tests de stress pour garantir la continuité de l'activité en cas de pic d'utilisation

Les équipes doivent donc suivre la pyramide des tests et mettre en œuvre un niveau de test approprié pour chacune des catégories de tests suivants.



- Les tests unitaires permettent de tester un composant (classe java), ou un bout de code, isolé de ses dépendances.
- Les tests d'intégration permettent de vérifier que tous les composants fonctionnent bien ensemble (composants internes et externes comme la base de données).
- Les tests d'acceptation et de bout en bout permettent de s'assurer que le système répond bien à l'ensemble des besoins fonctionnels.

S'agissant ici de tester une API via un POC pour une application qui n'est pas encore développée, on adaptera alors notre pyramide de test au contexte particulier des tests à effectuer pour une API.

A ces tests fonctionnels, s'ajoutent des tests non fonctionnels :tests de performance, de sécurité et de qualité.

4.2 Tests unitaires

La plus grosse section, à la base de la pyramide, est constituée des tests unitaires qui testent de "petites" unités de code. Plus précisément, ils testent que chaque fonctionnalité extraite de manière isolée se comporte comme attendu. Les tests unitaires permettent de tester un composant (classe java), ou un bout de code, isolé de ses dépendances.

Ils sont très rapides et faciles à exécuter. Si vous avez cassé quelque chose, vous pouvez le découvrir vite et tôt.

Les tests unitaires doivent suivre les principes F.I.R.S.T. Cela rend ces tests plus fiables, plus faciles à utiliser et, en fin de compte, plus bénéfiques pour toute l'équipe.

PRINCIPE	Qu'est-ce que cela signifie ?
Fast (Rapide)	Une demi-seconde, c'est déjà trop lent. Vous avez besoin que des milliers de tests soient exécutés en moins de quelques secondes.
Isolé et indépendant	Quand un test échoue, il vous faut savoir ce qui a échoué. Testez un élément à la fois et n'utilisez plus d'une assertion que s'il le faut vraiment.
Répétable	Vous ne pouvez pas vous fier aux tests qui échouent certaines fois et réussissent d'autres. Vous devez construire des tests qui n'interfèrent pas les uns avec les autres et qui sont assez autonomes pour donner toujours le même résultat.
Self-validating (Autovalidation)	Utilisez les bibliothèques d'assertions disponibles, écrivez des tests spécifiques, et laissez votre Framework de test s'occuper du reste. Vous pouvez vous y fier pour exécuter vos tests et générer des rapports de tests clairs.
Thourough (Approfondi)	Utilisez le TDD et écrivez vos tests en écrivant votre code. Explorez toute la gamme d'actions possibles de votre méthode et écrivez beaucoup de tests unitaires.

Pour notre API les tests unitaires vont correspondre à tester les composants individuellement soient les classes Java.

D'autre part, nous devons écrire nos tests unitaires à bon escient.

Outils : JUnit/Eclipse ou JUnit/Maven.

4.3 Tests d'intégration

Les tests d'intégration permettent de vérifier que tous les composants fonctionnent bien ensemble (composants internes et externes comme la base de données).

Nous devrions couvrir toute intégration avec des systèmes tiers que notre API utilise à ce stade de nos tests, soit avec une base de données, exemple PostgreSql.

Ici, la couverture du code n'a pas tant d'importance. Si l'exécution correcte de notre méthode dépend d'un système externe, nous devons tester si notre méthode est vraiment capable de se connecter à ce système et d'exécuter la méthode souhaitée de son côté.

Pour notre API les tests d'intégration vont correspondre à tester l'intégration de toutes les classes Java ainsi que les échanges avec la base de données PostgreSQL (CRUD). Si nous notre sous-système ERS fait appel à une API externe (géo localisation, calcul des distances, itinéraires), les tests d'intégration permettront également de s'assurer du bon fonctionnement de composant externe.

Outils :

- JUnit/(Eclipse ou Maven)
- Postman / Collections
- Github / Actions / Workflow / Pipeline CI/CD (fichier .yml).

4.4 Tests E2E

Ici, nous devrions tester toutes les fonctionnalités de notre API. La couverture du test E2E doit être basée sur la vérification du début à la fin des principales fonctionnalités métiers offertes par l'API.

Leur point principal devrait être de répondre à la question si, après avoir envoyé une requête à notre API, la réponse que nous obtenons répond aux conditions requises, et si tous les systèmes externes ont été mis à jour si besoin.

Les principaux objectifs des tests fonctionnels de l'API sont :

- pour s'assurer que l'implémentation fonctionne correctement comme prévu, pas de bugs !
- pour s'assurer que l'implémentation fonctionne comme spécifié conformément à la spécification des exigences (qui deviendra plus tard notre documentation API).
- pour éviter les régressions entre les fusions de code et les versions.

Ces tests fonctionnels seront décrits sous forme de scénarios utilisant la syntaxe Gherkin selon la méthode BDD.

Pour notre API les tests E2E vont correspondre à tester l'appel aux services de notre API depuis une interface utilisateur accessible au département de régulation des appels pour les urgences médicales.

Outils : Cuccumber ; Selenium …

4.5 Tests de performance

Les différents types de test de perfomance de l'API peuvent être automatisés avec les outils suivants : Gatling, JMeter.

4.5.1 Tests de performance

Les tests de performance tentent de tester le comportement du système dans des conditions de fonctionnement normales, sur la base d'estimations ou d'expériences antérieures.

Exemple de scénario :

> When 1000 users call the API within a timeframe of 180s seconds,
> the average response time should be below 250ms,
> the maxium response time should be below 500ms
> and no errors should occur.

4.5.2 Tests de charge

Les tests de charge tentent de tester le comportement du système dans des conditions élevées et supérieures à la normale.

Ces conditions peuvent simuler des pics de trafic temporaires que l'application doit pouvoir gérer.

Exemple de scénario :

> When 1000 users call the API within 30s seconds,
> the average response time should be below 1000ms
> and no errors should occur.

4.5.3 Tests de stress

Cette approche est une tentative de casser l'application en la submergeant de requêtes. L'application augmentera son utilisation du processeur et de la mémoire jusqu'à ce que quelque chose cesse de fonctionner.

En règle générale, les tests de résistance sont effectués en augmentant progressivement le nombre de demandes. L'un des résultats de ce type de test est de connaître la limite sous laquelle l'application fonctionne encore de manière acceptable.

Les tests de résistance tentent donc de tester le comportement du système dans des conditions extrêmes, bien au-dessus de ce qui devrait se produire dans des scénarios normaux. L'objectif étant de trouver le point de rupture afin de le gérer correctement.

4.6 Tests de sécurité

- **Sécurité et autorisation**

Vérifiez que l'API est conçue selon les principes de sécurité: refus par défaut, échec sécurisé, principe du moindre privilège, rejet de toutes les entrées illégales, etc.

Vérification positive : assurez-vous que l'API réponde correctement à l'autorisation via toutes les méthodes d'authentification convenues dans les spécifications.

Vérification négative: assurez-vous que l'API refuse tous les appels non autorisés.

Autorisations de rôle : assurez-vous que des points de terminaison spécifiques sont exposés à l'utilisateur en fonction de son rôle. L'API doit refuser les appels aux points de terminaison qui ne sont pas autorisés pour le rôle de l'utilisateur.

Protocole : vérifier HTTP/HTTPS conformément aux spécifications.

Fuites de données : assurez-vous que les représentations de données internes qui doivent rester internes ne fuient pas vers l'API publique lors du chargement de la réponse.

Politiques de limitation de débit, d'étranglement et de contrôle d'accès.

- **Les missions du SSI**

Mise en place de scans de vulnérabilités internes et externes afin d'analyser les résultats pour en déterminer un plan d'actions et permettre aux différentes intervenants d'effectuer les corrections.

Mise en place de tests d'intrusion dans le but de réaliser un plan d'actions pour remédier aux failles détectées.

Mise en place des audits du trafic réseau pour améliorer la disponibilité du SI.

Mise en place d'un service « computer emergency response team » (CERT, centre d'alerte et de réaction aux attaques informatiques), permettant de disposer d'un prestataire de réponse à incident en 24×7 pour limiter l'impact d'une attaque et aider à rétablir le système d'information.

Mise en place d'exercices de gestion de cyberattaque, de panne du système pour identifier les axes d'amélioration à la gestion de la crise, de la documentation, de la continuité d'activité etc.

- RGPD

S'assure la sécurité des données et leur confidentialité.

S'assurer que toutes les données du patient sont correctement protégées.

4.7 Test de qualité

La surveillance est indispensable, surtout dans le cadre d'une architecture micro-services. Il s'agit de surveiller les tests fonctionnels et non fonctionnels (performance, sécurité) et également la qualité du code Java.

La surveillance (Monitoring) fait partie intégrante de l'assurance de la qualité de notre API. Avec les tests de performances, nous pouvons nous fournir de nombreuses informations sur le comportement de notre environnement et sur le nombre de ressources, telles que le processeur et la RAM, que notre API utilisera en production. Ces informations peuvent nous aider à préparer notre infrastructure d'environnement de production, nous épargnant ainsi beaucoup de problèmes.

Outils : Jacoco ; SonarCloud

4.8 Critères d'échelonnement

Nous décrivons dans ce paragraphe les critères d'échelonnement généraux cependant les critères exacts devront être établis durant le projet même. Si aucun autre accord ou exigence n'est spécifié, les critères ci-dessous sont applicables.

Bogues

Il ne peut y avoir de bogues présentant le degré de gravité « Majeur » ou « Bloquant ».

Il doit subsister moins de 5 bogues non résolues de gravité « Moyenne » et ceux-ci sont connus et acceptés.

Il doit subsister moins de 20 bogues non résolues de gravité « Mineur » et ceux-ci sont connus et acceptés.

Degré de couverture

Pour la validation en production, l'objectif est de couvrir chaque élément de spécification, y compris les mécanismes de sécurité.

Le taux de couverture des exigences fonctionnelles doit être à 100%.

Le taux de couverture des exigences non fonctionnelles doit être à 100%.

Le taux de réussite de tous les tests manuels et automatiques doit être de 100%.

Remarque pour la validation du POC de l'API ERS, l'architecture technique cible n'ayant pas été finalisée, le taux de couverture des mécanismes de sécurité ne sera de ce faite, pas appliqué.

5 Outils de test

Les outils de test vont permettre d'automatiser les tests, simuler l'opération de l'utilisateur et générer les résultats des tests.

Outil	Activités	Remarque
JUnit 5	Tests unitaires et intégration	Framework de test unitaire pour Java. Chaque composant sera testé avec des cas de test (TestCase). Un TestCase permettant tester le bon fonctionnement d'une classe. Junit / AssertJ (gère critère acceptation, langage compréhensible)
Postman	Tests intégration Tests de performance	Permet de tester manuellement les appels à l'API via des requêtes HTTP(Get, Post). Ces différents appels peuvent être enregistré dans des collections pour être relancés ultérieurement. Postman permet d'ajouter des tests écrits en javascript.
Github Actions	Tests intégration Pipeline CI/CD	Github Actions permet de gérer l'intégration et le déploiement continue en créant un pipeline CI/CD avec la création d'un Workflow via un fichier yaml.
SonarCloud	Tests de qualité	JaCoCo (check couverture de test - qualité) SonarCloud (check qualité code)
Cucumber	Spécification des tests Tests E2E	Framework pour l'implémentation des scénarios de Type BDD, écrits en Gherkin afin de permettre l'automatisation de tests. Compatible avec plusieurs langages de programmation dont JS (Cucumber.JS). Génère des rapports (HTML, JSON) sur les scénarios de test (passed, failed).
Selenium	Automatisation des tests navigateurs web Tests E2E	Selenium-WebDriver Fournit des extensions afin d'émuler des interactions utilisateur avec les navigateurs, un serveur de distribution permettant la mise à l'échelle de l'allocation de navigateur ainsi que l'infrastructure pour l'implémentation de la spécification W3C..
Gatling JMeter	Tests performance	Outil open-source de test de charge et de performance pour applications web
Jira	Gestion des anomalies	Permet la gestion collaborative des anomalies suites aux tests. Intégration est transparente entre Jira et GitHub Gestion de la documentation à jour via Git repository.