**UNIVERSITÉ DU QUÉBEC À MONTRÉAL**

**DÉVELOPPEMENT ET VALIDATION D’UN FRAMEWORK D’AUTOMATISATION DES TESTS POUR API REST**

**MÉMOIRE PRÉSENTÉ COMME EXIGENCE PARTIELLE**

**DU D.E.S.S EN GÉNIE LOGICIEL**

**PAR**

**MOHAMED JLASSI**

**DECEMBRE 2024**

# **REMERCIEMENTS**

Je tiens à exprimer ma sincère gratitude à mon tuteur, **M. Hafedh Mili**, pour son accompagnement et son soutien tout au long de ce projet. Ses conseils précieux, sa disponibilité et son expertise ont grandement contribué à la réussite de ce travail.

Je remercie également mes proches pour leur encouragement infaillible et leur compréhension durant cette période exigeante. Leur soutien moral a été une source précieuse de motivation et de sérénité.

Enfin, je souhaite remercier toutes les personnes qui, de près ou de loin, ont contribué à enrichir cette expérience, que ce soit par leurs retours, leurs discussions ou leur inspiration.

À tous, je vous adresse mes remerciements les plus sincères.

Table des matières

[REMERCIEMENTS 2](#_Toc185972732)

[RÉSUMÉ 6](#_Toc185972733)

[INTRODUCTION 7](#_Toc185972734)

[CHAPITRE 1 : PROJET D'AUTOMATISATION DES TESTS D'API 11](#_Toc185972735)

[1.1 Contexte 11](#_Toc185972736)

[**1.1.1.** **Problématique** 12](#_Toc185972737)

[**1.1.2 Solution** 13](#_Toc185972738)

[1.2. Missions 16](#_Toc185972739)

[**1.2.1 La portée** 16](#_Toc185972740)

[**1.2.2 Les objectifs** 17](#_Toc185972741)

[**1.2.3 Les enjeux** 17](#_Toc185972742)

[CHAPITRE 2: ÉTAT DE L'ART 20](#_Toc185972743)

[2.1 Cadre Général 20](#_Toc185972744)

[2.2. Aperçu des outils existants 20](#_Toc185972745)

[**2.2.1. Bibliothèques et outils open source** 20](#_Toc185972746)

[**2.2.2 Solutions commerciales** 23](#_Toc185972747)

[2.3 Architecture du Framework Développé 24](#_Toc185972748)

[**2.3.1 Vue d’Ensemble Technique** 24](#_Toc185972749)

[**2.3.2 Design Pattern Page Object Model (POM)** 25](#_Toc185972750)

[**2.3.3 Tests Basés sur des Données (Data-Driven Testing)** 25](#_Toc185972751)

[2.4. Data-Driven Testing : Mise en Œuvre et Exemple Pratique 26](#_Toc185972752)

[**2.4.1. Gestion des Secrets** 30](#_Toc185972753)

[**2.4.2. Génération de Rapports** 31](#_Toc185972754)

[2.5 Positionnement par rapport aux Outils Existants 32](#_Toc185972755)

[CHAPITRE 3 : MÉTHODOLOGIE 34](#_Toc185972756)

[3.1 Approche Kanban 34](#_Toc185972757)

[3.2 Mise en Place 35](#_Toc185972758)

[**3.2.1 Structure Maven** 35](#_Toc185972759)

[3.3 Développement 36](#_Toc185972760)

[**3.3.1 Préparation** 36](#_Toc185972761)

[**3.3.2 Qualité du Code** 37](#_Toc185972762)

[**3.3.3 Architecture Modulaire** 37](#_Toc185972763)

[CHAPITRE 4 : RÉSULTATS 39](#_Toc185972764)

[4.1 Fonctionnalités Testées 39](#_Toc185972765)

[4.2 Performances 40](#_Toc185972766)

[4.3 Tests Basés sur les Données (Data-Driven Testing) 40](#_Toc185972767)

[4.4 Rapports Générés 41](#_Toc185972768)

[4.5 Exécution et Intégration CI/CD 42](#_Toc185972769)

[CHAPITRE 5 : DISCUSSION 43](#_Toc185972770)

[5.1 Rétrospective 43](#_Toc185972771)

[**5.1.1 Points Positifs** 43](#_Toc185972772)

[**5.1.2 Points Négatifs** 44](#_Toc185972773)

[5.2 Obstacles Rencontrés 44](#_Toc185972774)

[5.3 Perspectives 45](#_Toc185972775)

[**5.3.1 Améliorations Techniques** 45](#_Toc185972776)

[**5.3.2 Améliorations Collaboratives** 46](#_Toc185972777)

[**5.3.3 Extensions Fonctionnelles** 46](#_Toc185972778)

[CHAPITRE 6 : CONCLUSION 47](#_Toc185972779)

[Bibliographie 48](#_Toc185972780)

**Liste des figures**

[Figure 1 : Exemple de structure data-manager.json 27](#_Toc185972830)

[Figure 2 : Exemple de logs 28](#_Toc185972831)

[Figure 3 : Execution d'un suite de test 29](#_Toc185972832)

[Figure 4 : Rapport : Tous les tests réussis Capture d’écran affichant le résultat de l’exécution où tous les tests ont été validés avec succès. 31](#_Toc185972833)

[Figure 5: Rapport : Échec d’un test avec détails Capture d’écran montrant un test échoué, avec une distinction claire des tests réussis, échoués, et ignorés. 32](#_Toc185972834)

[Figure 6 : Rapport : Différence entre les valeurs actuelles et attendues Capture d’écran détaillant l’échec d’un test, indiquant les logs, la valeur actuelle (actual value) et la valeur attendue (expected value). 32](#_Toc185972835)

[Figure 7 : Exemple de stub pour un endpoint simulé 36](#_Toc185972836)

**Liste des abréviations, sigles et acronymes**

**API** : Application Programming Interface

**CI/CD** : Continuous Integration / Continuous Deployment

**DDT** : Data-Driven Testing

**JSON** : JavaScript Object Notation

**POM** : Page Object Model

**POJO** : Plain Old Java Object

**JWT** : JSON Web Token

**REST** : Representational State Transfer

**WireMock** : Outil de simulation pour stubber les API REST

**TestNG** : Test Next Generation, framework de gestion des tests en Java

**Maven** : Outil de gestion des dépendances et de construction de projets Java

**JIRA** : Outil de gestion des projets et des tâches

**Xray** : Extension JIRA pour la gestion des tests

**AWS** : Amazon Web Services

**CLI** : Command Line Interface

# **RÉSUMÉ**

Les API REST sont devenues un élément central des systèmes modernes, assurant la communication entre services distribués. Toutefois, leur validation manuelle, souvent répétitive et chronophage, pose des défis majeurs en termes d’efficacité, de couverture, et de fiabilité. Ce projet propose un **framework générique de tests automatisés pour API REST**, conçu pour surmonter ces défis grâce à une architecture modulaire, évolutive et sécurisée.

Reposant sur des technologies open source telles que **RestAssured** et **TestNG**, ce framework intègre des fonctionnalités avancées adaptées aux besoins des développeurs et des équipes de test :

1. **La gestion des secrets**, garantissant la sécurité des données sensibles (par exemple, jetons JWT, clés API) à travers des fichiers encryptés ou des gestionnaires externes.
2. **Le data-driven testing**, qui permet de générer des scénarios variés en s’appuyant sur des fichiers JSON pour configurer dynamiquement les cas de test.
3. **L’architecture Page Object Model (POM)**, qui encapsule les endpoints API dans des objets réutilisables, facilitant ainsi la maintenance et la lisibilité du code.
4. **L’intégration CI/CD**, permettant une exécution automatique et continue des tests dans des pipelines tels que Jenkins ou GitHub Actions.

Afin d’illustrer ses capacités, le projet intègre un exemple concret basé sur l’application *Profile Management*. Dans ce contexte, **WireMock** est utilisé pour simuler les réponses des dépendances externes des API. Cette approche permet de valider le framework dans un environnement contrôlé et reproductible. Les résultats obtenus démontrent une réduction notable du temps de test, une couverture exhaustive des cas critiques, et une meilleure gestion des scénarios complexes.

# **INTRODUCTION**

Dans un contexte où les API REST sont omniprésentes, les équipes de développement et de test sont confrontées à la nécessité de valider rapidement et efficacement des fonctionnalités souvent complexes et interconnectées. Tester ces API manuellement est non seulement long et répétitif, mais également sujet aux erreurs humaines, surtout lorsque les systèmes évoluent rapidement ou intègrent des dépendances externes.

Pour répondre à ces défis, ce projet propose un **framework générique et automatisé** spécifiquement conçu pour les API REST. Contrairement aux outils existants tels que Postman ou SoapUI, souvent limités à des scénarios spécifiques ou difficiles à intégrer dans des pipelines CI/CD, ce framework offre une solution complète et évolutive adaptée à divers besoins professionnels.

* **Objectifs principaux du framework**

Le framework a été conçu pour répondre aux exigences suivantes :

1. **Automatisation universelle des tests API REST** :
   * Support de toutes les méthodes HTTP (GET, POST, PUT, DELETE).
   * Validation fonctionnelle, de régression et des données retournées (par exemple, schéma JSON et statuts HTTP).
2. **Sécurité des données sensibles** :
   * Intégration d’une gestion des secrets pour protéger les informations sensibles, comme les clés API ou les jetons JWT.
   * Utilisation de fichiers encryptés ou d’outils externes (par exemple, Vault) pour une sécurité renforcée.
3. **Flexibilité grâce au data-driven testing** :
   * Utilisation de fichiers JSON pour configurer dynamiquement les cas de test, permettant une variation facile des entrées et des scénarios sans modifier le code.
4. **Modularité et maintenabilité avec le design pattern POM** :
   * Encapsulation de chaque endpoint dans des classes spécifiques, rendant les tests plus lisibles, réutilisables et faciles à maintenir.

**Illustration par un cas d’usage : Profile Management**

Afin de démontrer l’efficacité du framework, le projet intègre l’exemple de l’application *Profile Management*. Cette application gère des opérations CRUD (Create, Read, Update, Delete) sur des profils utilisateurs. Pour simuler les dépendances externes, le projet utilise **WireMock**, permettant de créer un environnement isolé et contrôlé où les réponses des API externes sont simulées. Cette approche assure que les tests du framework peuvent être validés indépendamment des fluctuations des services externes.

* **Fonctionnalités principales du framework**

1. **Gestion des secrets**

La sécurité des données est essentielle dans tout projet impliquant des API REST. Ce framework inclut un système de gestion des secrets qui :

* + Stocke et protège les données sensibles (clés, mots de passe, jetons) dans des fichiers encryptés.
  + S’intègre facilement à des gestionnaires externes comme HashiCorp Vault pour les grandes entreprises.
  + Offre une flexibilité pour configurer des environnements distincts (par exemple, développement, test, production) sans compromettre la sécurité.

1. **Data-driven testing**

Le framework repose sur une approche orientée données, où les scénarios de test sont générés dynamiquement à partir de fichiers JSON. Cette méthode offre plusieurs avantages :

* + Réduction de la duplication de code en centralisant les données d’entrée.
  + Facilité de mise à jour : ajouter un nouveau scénario ne nécessite qu’une modification du fichier JSON correspondant.
  + Adaptabilité pour tester divers cas d’utilisation, y compris les réponses valides et les erreurs.

1. **Architecture Page Object Model (POM)**

Inspiré des bonnes pratiques en automatisation, le design pattern POM structure le framework en encapsulant chaque endpoint dans une classe dédiée. Par exemple :

* + La classe UserProfileAPI gère toutes les opérations relatives aux profils utilisateurs.
  + Les méthodes telles que createUser(), getUser(), ou deleteUser() sont réutilisables dans différents scénarios de test.  
    Cette organisation rend le code facile à lire, à étendre et à maintenir.

1. **Intégration CI/CD**

Le framework est conçu pour être facilement intégré dans des pipelines d’intégration continue. Cela permet une exécution automatisée des tests après chaque déploiement ou mise à jour, garantissant une qualité constante.

Ce projet représente une avancée significative dans l’automatisation des tests pour API REST. En combinant sécurité, modularité et flexibilité, le framework offre une solution robuste et économique pour valider les fonctionnalités critiques dans des environnements modernes. L’intégration de fonctionnalités avancées comme la gestion des secrets et les tests data-driven, associée à l’exemple concret de *Profile Management*, illustre sa capacité à répondre aux besoins réels des développeurs et des entreprises.

# **CHAPITRE 1 : PROJET D'AUTOMATISATION DES TESTS D'API**

## **Contexte**

Dans les systèmes modernes, les API REST constituent la pierre angulaire des interactions entre différents services logiciels. Elles permettent une communication fluide entre des composants souvent développés avec des technologies variées et déployés sur des environnements distribués. Cependant, cette interconnexion complexe engendre de nombreux défis, notamment lors de l’ajout de nouvelles fonctionnalités ou de la modification des API existantes.

Un des principaux défis réside dans la **fragilité des interconnexions**. Une modification ou mise à jour d’une API peut provoquer des dysfonctionnements imprévus sur les systèmes qui y dépendent, en aval ou en amont. Par exemple, lorsqu’une nouvelle fonctionnalité est introduite, elle doit non seulement répondre aux besoins spécifiques, mais aussi préserver les interactions existantes. Cela nécessite une validation rigoureuse des comportements attendus à chaque étape.

Un autre défi majeur est lié à la **multiplicité des scénarios de test**. Les API modernes, en raison de leur flexibilité, doivent gérer divers paramètres d’entrée, produire des réponses adaptées et répondre à différents types d’utilisateurs ou de rôles. Cette flexibilité entraîne une explosion combinatoire des cas de test possibles. Par exemple, pour une API d’authentification, les scénarios peuvent inclure des tests pour plusieurs rôles d’utilisateur, différents types de tokens, et des conditions spécifiques comme des tokens expirés ou invalides. Cette complexité rend les approches manuelles insuffisantes pour garantir une couverture complète.

Dans les **environnements distribués**, les dépendances externes ajoutent un niveau supplémentaire de complexité. Les services tiers, bases de données ou autres API externes ne sont pas toujours disponibles ou fiables dans les environnements de test. Ces limitations rendent difficile la validation des scénarios impliquant des interactions dynamiques ou des dépendances complexes.

Enfin, la **sécurité des données sensibles** est une préoccupation cruciale. Les tests nécessitent souvent l’utilisation de clés API, de jetons d’authentification ou d’autres informations sensibles. Ces données doivent être protégées contre toute divulgation accidentelle, ce qui complique davantage leur gestion dans des environnements de développement ou de test.

Face à ces défis, il devient impératif d'adopter un cadre structuré pour les tests automatisés, permettant une validation efficace des API tout en répondant aux exigences modernes en termes de modularité, de sécurité et d’intégration dans des pipelines de développement continu.

### **Problématique**

Dans un contexte où les API REST sont devenues indispensables pour assurer la communication entre des systèmes distribués, leur validation reste un défi important. Les environnements logiciels modernes sont caractérisés par des cycles de développement courts, une forte interdépendance entre les services et une nécessité croissante d’automatiser les tâches pour garantir la qualité des livraisons. Les tests manuels, bien qu’utiles pour certaines validations ponctuelles, s’avèrent insuffisants pour répondre à ces besoins.

Un des principaux défis réside dans la **complexité des scénarios de test**. Les API doivent gérer une multitude de cas d’utilisation, chacun nécessitant des combinaisons spécifiques de paramètres d’entrée et de conditions de réponse. Tester toutes les possibilités devient rapidement impraticable sans une automatisation robuste. Par exemple, une API d’authentification peut impliquer des scénarios variés : requêtes valides avec des jetons d’accès corrects, tentatives avec des jetons expirés, ou encore accès non autorisés pour des rôles inadéquats. Les tests manuels ne peuvent couvrir efficacement cette diversité.

Par ailleurs, les méthodologies modernes comme **l’Agilité** et les pratiques **DevOps** imposent un rythme accéléré de développement et de livraison. Dans ce cadre, les tests doivent être intégrés en continu dans le pipeline de développement. Cette philosophie de **"Shift Left Testing"**, qui prône une intervention précoce des tests dans le cycle de développement, nécessite de disposer de tests automatisés prêts à valider chaque nouvelle modification dès son introduction. Cela garantit que les défauts sont identifiés rapidement, réduisant ainsi les coûts et les retards associés aux corrections tardives.

En parallèle, la pratique de la **livraison par petits lots**, commune dans les environnements DevOps, nécessite des validations rapides et fiables de chaque fonctionnalité avant son déploiement. Une simple mise à jour peut impacter l’ensemble du système si elle n’est pas correctement testée, ce qui renforce l’importance d’un framework capable de s’adapter à des tests fréquents et ciblés.

Enfin, la **sécurité des données sensibles** est une problématique incontournable. Les API nécessitent souvent l’utilisation de jetons, clés ou autres informations sensibles pour fonctionner. Ces éléments doivent être protégés tout en restant accessibles dans des environnements de test, sous peine de compromettre la sécurité ou de compliquer l’exécution des scénarios.

Ces défis soulignent la nécessité d’une solution moderne, capable de :

* Automatiser les tests pour réduire les efforts manuels et couvrir une large gamme de scénarios.
* S’intégrer dans des pipelines CI/CD pour valider en continu les modifications apportées aux API.
* Simuler efficacement les dépendances externes afin de garantir des tests stables et reproductibles.
* Gérer de manière sécurisée les secrets et données sensibles pour limiter les risques de fuite.
* Soutenir les approches Agiles et DevOps en permettant des livraisons fréquentes et des validations rapides, alignées avec le principe du "Shift Left".

Le développement d’un framework répondant à ces exigences représente une réponse aux besoins des environnements modernes, où la rapidité, la sécurité et la qualité sont des impératifs. Ce projet vise précisément à combler ces lacunes, en proposant une solution adaptée aux réalités des tests d’API REST.

### **1.1.2 Solution**

Le framework d’automatisation des tests API développé dans ce projet vise à offrir une **solution moderne et robuste** pour valider efficacement la qualité des API REST. Face aux défis croissants liés à la complexité des systèmes distribués et aux besoins d’intégration continue, ce framework propose une approche modulaire et orientée développeur, simplifiant les processus de test tout en assurant une couverture optimale.

Les principaux objectifs de cette solution sont les suivants :

1. **Automatiser les tests des API** : Fournir un outil performant pour exécuter des tests fonctionnels, de régression et de scénarios d’erreur de manière rapide et fiable.
2. **Faciliter la maintenance et l’évolutivité** : Grâce à une architecture basée sur le **design pattern Page Object Model (POM)**, le framework garantit une séparation claire entre la logique des tests, les données et les interactions API.
3. **Assurer une sécurité renforcée** : Intégrer une gestion centralisée et dynamique des secrets via des variables d’environnement pour protéger les informations sensibles comme les clés API et tokens d'authentification.
4. **Optimiser l’intégration CI/CD** : Permettre une exécution fluide et automatisée des tests dans les pipelines d’intégration continue, avec des rapports détaillés pour un suivi simplifié.
5. **Adapter les tests à différents environnements** : Offrir une flexibilité dans la configuration des données grâce à des fichiers **JSON** organisés par environnement (développement, test, production).

#### **1.1.2.1 Composants clés du framework**

1. **Structure modulaire et réutilisable**

Le framework est conçu pour isoler les différents aspects des tests dans des modules indépendants, facilitant ainsi l’ajout de nouveaux tests et la modification des scénarios existants.

1. **Data-Driven Testing**

Les tests sont entièrement basés sur des données externes, définies dans des fichiers **JSON**, permettant de créer des scénarios variés sans modifier le code source.

1. **Validation robuste des API**

Chaque réponse d’API est désérialisée en objets Java (POJO) et comparée aux résultats attendus, garantissant une validation précise des attributs et des valeurs retournées.

1. **Rapports exploitables**

Les rapports générés fournissent des informations détaillées sur les résultats des tests, intégrant les succès, échecs, erreurs et logs associés, tout en étant compatibles avec les outils CI/CD.

1. **Gestion centralisée des secrets**

Les secrets, comme les clés API et tokens, sont stockés de manière sécurisée et injectés dynamiquement via des variables d’environnement pour garantir leur confidentialité.

#### **1.1.2.2. Bénéfices pour les équipes de développement et de test**

* **Facilité d’utilisation** : Une organisation intuitive et un fonctionnement plug-and-play permettent une adoption rapide par les développeurs et les ingénieurs en test.
* **Réduction des erreurs humaines** : Grâce à l’automatisation, les scénarios critiques sont couverts de manière systématique, éliminant les risques liés aux tests manuels.
* **Efficacité opérationnelle** : Le framework permet une exécution rapide et une analyse simplifiée des résultats, accélérant le cycle de développement logiciel.

Cette solution se distingue par sa capacité à s’adapter aux besoins des équipes tout en maintenant une simplicité d’utilisation et une efficacité optimale. Elle constitue un outil essentiel pour garantir la qualité et la stabilité des API dans des environnements logiciels modernes.

## **Missions**

Le framework a été conçu pour résoudre les défis complexes associés aux tests des API REST dans des environnements logiciels modernes. Sa mission principale est de simplifier, standardiser et sécuriser les processus de validation, en offrant une solution efficace et accessible pour les développeurs et les ingénieurs en automatisation de tests (**SDET**). Il répond aux besoins d’une automatisation fiable tout en permettant une maintenance aisée et une extension rapide.

### **1.2.1 La portée**

La portée de ce projet consiste à développer un **framework d’automatisation des tests API** performant, modulaire et adaptable, capable de répondre aux besoins des environnements logiciels modernes.

* **Automatisation complète des tests** : Couvre les **tests fonctionnels**, les **scénarios d’erreur** et les **tests de régression** pour assurer la conformité et la stabilité des API.
* **Architecture modulaire et réutilisable** : Facilite la création, l’exécution et la maintenance des tests grâce à des composants structurés et une organisation claire des données de test via des fichiers **JSON**.
* **Intégration CI/CD** : Assure une exécution fluide et automatisée dans les pipelines d'intégration continue, permettant une validation rapide et continue des nouvelles fonctionnalités.
* **Gestion sécurisée des secrets** : Centralise et sécurise les données sensibles avec une injection dynamique via des variables d’environnement pour protéger les informations critiques.
* **Flexibilité et évolutivité** : Simplifie l’ajout de nouveaux tests grâce à une organisation intuitive et une structure **plug-and-play**.
* **Soutien aux méthodologies modernes** : Facilite l’adoption du **Shift Left Testing** et de la livraison itérative pour détecter les anomalies plus tôt dans le cycle de développement.
* **Limitation principale** : Le framework ne prend pas en charge les **tests non fonctionnels**, tels que les tests de performance ou de sécurité, qui nécessitent des outils spécialisés comme **JMeter** ou **OWASP ZAP** pour compléter la stratégie de test.

Ce framework est conçu pour offrir une **simplicité d’utilisation**, une **efficacité opérationnelle**, et une **adaptabilité**, tout en se concentrant sur la qualité fonctionnelle des API dans des environnements logiciels variés.

### **1.2.2 Les objectifs**

Le développement de ce framework s’articule autour de plusieurs objectifs clés :

1. **Rendre les tests API entièrement automatisés** pour améliorer la fiabilité et réduire les dépendances aux validations manuelles.
2. **Permettre une adoption rapide** grâce à une structure conviviale et facile à comprendre, même pour les utilisateurs non experts.
3. **Accélérer l’ajout de nouveaux tests** en utilisant des modèles génériques et réutilisables.
4. **Maximiser la couverture des tests** en intégrant des scénarios variés, y compris les cas d’erreur et les cas limites.
5. **Simplifier la maintenance des tests existants** en limitant les duplications et en isolant les modifications à des composants spécifiques.
6. **Garantir la sécurité des données sensibles**, telles que les clés d’accès et les jetons, en adoptant des solutions de gestion sécurisée.
7. **Soutenir les pratiques Agiles et DevOps** en intégrant des tests continus et en permettant des validations rapides et fréquentes des nouvelles fonctionnalités.

### **1.2.3 Les enjeux**

La mise en place et l’utilisation de ce framework impliquent plusieurs défis majeurs qui doivent être relevés :

1. **Gestion de la complexité des tests API**

Les API modernes présentent une variété de scénarios complexes, allant des flux de données dynamiques aux interactions avec des services tiers. Le framework doit permettre une validation exhaustive sans ajouter de complexité inutile.

1. **Simplicité et rapidité d’utilisation**

Les tests doivent être faciles à configurer et à exécuter, même pour les équipes ayant peu d’expérience dans l’automatisation. Le design du framework doit minimiser les efforts initiaux et favoriser une prise en main rapide.

1. **Facilité de maintenance**

Avec l’évolution continue des API, la capacité à mettre à jour les tests de manière localisée et sans impact global est cruciale. Le framework doit offrir une modularité permettant une maintenance efficace et prévisible.

1. **Sécurisation des données sensibles**

Les tests impliquent souvent l’utilisation d’informations critiques, telles que des clés API ou des jetons JWT. La gestion de ces secrets doit être sécurisée tout en permettant leur utilisation fluide dans différents environnements.

1. **Adaptabilité aux cycles rapides de développement**

Les pratiques modernes, comme l’intégration continue et le déploiement continu, nécessitent des tests rapides et fiables pour accompagner des livraisons fréquentes. Le framework doit pouvoir s’intégrer harmonieusement dans ces cycles.

1. **Extension et évolutivité**

Le framework doit être conçu pour évoluer avec les besoins des projets, permettant l’ajout de nouveaux endpoints ou de nouveaux cas de test sans réorganiser l’ensemble de la structure.

1. **Respect des méthodologies modernes**

En soutenant des pratiques telles que le **Shift Left Testing**, le framework doit permettre une validation précoce et continue, tout en favorisant une livraison rapide par petits lots.

Ce framework répond aux attentes des environnements logiciels modernes en combinant simplicité, flexibilité et sécurité. En réduisant la complexité des tests API et en facilitant leur maintenance, il constitue une solution essentielle pour les équipes techniques cherchant à maximiser la qualité et la rapidité de leurs livraisons.

# **CHAPITRE 2: ÉTAT DE L'ART**

## **2.1 Cadre Général**

Dans le développement logiciel moderne, les API REST sont des éléments clés pour connecter des services et des systèmes, offrant une interopérabilité entre des applications hétérogènes. Cependant, cette interconnectivité accrue introduit des défis spécifiques, tels que :

* **La gestion des dépendances** : Assurer une validation fonctionnelle même en l'absence de services tiers ou en cas de modifications d'API existantes.
* **L’intégration CI/CD** : Permettre une validation continue et rapide dans les pipelines d’intégration continue.
* **La sécurité des secrets** : Garantir la protection des clés API, des tokens et autres données sensibles.

Pour répondre à ces enjeux, les frameworks de tests d’API REST jouent un rôle essentiel en assurant la qualité et la stabilité des services. Ce chapitre examine les outils existants, leurs forces et limites, tout en positionnant le framework développé comme une solution adaptée aux besoins actuels.

## **2.2. Aperçu des outils existants**

### **2.2.1. Bibliothèques et outils open source**

**1. Postman/Newman**

* + **Description :** est une bibliothèque Java open-source conçue pour automatiser les tests des API REST.
* **Forces :**
  + Syntaxe fluide et lisible pour simplifier l'écriture des tests.
  + Prise en charge des formats JSON/XML et des authentifications complexes (Basic, OAuth, Bearer).
  + Intégration facile avec TestNG et JUnit.
* **Limites :**
  + Exclusivement pour les projets Java.
  + Nécessite des outils externes pour générer des rapports détaillés.
  + Courbe d’apprentissage pour les débutants.

**2. Postman/Newman**

* **Description :**Postman est une interface graphique intuitive pour tester manuellement les API REST, tandis que Newman permet d'exécuter des collections Postman via CLI pour l'automatisation des tests dans des pipelines CI/CD.
* **Forces :**
  + Interface conviviale et intuitive pour les tests manuels.
  + Facilité de création, d’organisation et de partage de collections de tests.
  + Compatible avec Newman pour l'intégration dans les pipelines CI/CD.
* **Limites :**
  + Les secrets (comme les clés API et tokens) sont stockés dans les bases de données Postman, soulevant des inquiétudes en matière de sécurité.
  + Peu adapté aux tests complexes ou à grande échelle nécessitant des validations avancées.

**3. Thunder Client**

* **Description :**Une extension légère pour Visual Studio Code conçue pour tester rapidement des endpoints API sans quitter l'environnement de développement.
* **Forces :**
  + Facilité d’utilisation pour des tests ponctuels dans Visual Studio Code.
  + Interface minimaliste idéale pour les développeurs.
* **Limites :**
  + Absence de support pour l’intégration dans les pipelines CI/CD.
  + Fonctionnalités limitées pour les scénarios de test complexes ou les tests data-driven.

***Source* :** (Bhatra, 2024)

**4. Karate Framework**

* **Description :**Karate est un framework de test open-source combinant tests API, UI, et performance, avec une syntaxe lisible basée sur Gherkin.
* **Forces :**
  + Intégration naturelle des tests BDD (Behavior-Driven Development).
  + Capacité à tester simultanément les API et les interfaces utilisateur.
* **Limites :**
  + Intégration dans des pipelines CI/CD complexes nécessitant des configurations supplémentaires.
  + Moins adapté aux projets nécessitant une forte modularité ou une réutilisation intensive des composants.

***Source* : (**Karate vs. Postman, 2023)

### **2.2.2 Solutions commerciales**

**1. ReadyAPI (anciennement SoapUI Pro)**

* **Description :** ReadyAPI est une suite d'outils premium développée par SmartBear pour les tests fonctionnels, de performance et de sécurité des API REST et SOAP. Elle offre des fonctionnalités avancées pour la création, l'exécution et la gestion des tests d'API, surpassant les capacités de SoapUI Open Source.
* **Limites :**
  + **Complexité pour les débutants :** L'étendue des fonctionnalités de ReadyAPI peut représenter une courbe d'apprentissage significative pour les nouveaux utilisateurs, nécessitant une formation ou une expérience préalable en tests d'API.
  + **Coût élevé** : En tant que solution commerciale, ReadyAPI implique des frais de licence qui peuvent être substantiels, particulièrement pour les petites équipes ou les projets avec des budgets limités.

***Source*** : (AbnAsia.org, 2024)

**2. Katalon Studio**

* **Description :** Katalon Studio est une plateforme d'automatisation de tests qui prend en charge les tests d'API, d'applications web, mobiles et de bureau. Elle est reconnue pour son interface conviviale et ses fonctionnalités intégrées, facilitant la création et l'exécution de tests automatisés.
* **Limites :**
  + **Fonctionnalités avancées accessibles uniquement via un abonnement** : Bien que Katalon Studio offre une version gratuite avec des fonctionnalités de base, certaines fonctionnalités avancées, telles que des options de personnalisation étendues et des intégrations spécifiques, sont réservées à la version Enterprise, nécessitant un abonnement payant.
  + **Problèmes de performance :** Certains utilisateurs ont signalé des ralentissements ou des blocages occasionnels lors de l'utilisation de Katalon Studio, ce qui peut affecter l'efficacité des tests.

Ces solutions commerciales offrent des fonctionnalités robustes pour l'automatisation des tests d'API, mais il est essentiel de considérer leurs limitations et coûts associés lors du choix de l'outil le mieux adapté aux besoins spécifiques de votre projet.

***Source*** : (Karate vs. Katalon, 2024)

## **2.3 Architecture du Framework Développé**

### **2.3.1 Vue d’Ensemble Technique**

Le framework développé repose sur une architecture modulaire, conçue pour maximiser la réutilisabilité et la simplicité tout en s’intégrant parfaitement dans les pipelines CI/CD modernes.

**Caractéristiques principales :**

* **Langage principal :** Java, choisi pour sa robustesse, sa popularité dans les environnements professionnels, et sa compatibilité avec des bibliothèques comme RestAssured pour les appels API et TestNG pour la gestion des suites de tests.
* **Gestion des dépendances :** Maven est utilisé pour centraliser et organiser les bibliothèques et plugins nécessaires, garantissant une configuration standardisée et reproductible.
* **Structure modulaire** : Basée sur le design pattern Page Object Model (POM), qui favorise une organisation claire et maintenable.

### **2.3.2 Design Pattern Page Object Model (POM)**

Le POM est au cœur de l'architecture, structurant le framework autour de la séparation des responsabilités pour assurer la maintenabilité et la réutilisabilité.

**Composants clés :**

* **Logique des tests :** Les scénarios de test sont encapsulés dans des classes dédiées, garantissant une organisation cohérente et centralisée.
* **Données d’entrée :** Stockées dans des fichiers JSON, elles sont indépendantes du code, facilitant l’ajout ou la modification de cas de test.
* **Modèles de données (POJOs) :** Sérialisent et désérialisent les réponses des API, permettant de valider les attributs et valeurs attendus en les comparant directement avec les données des fichiers JSON.

**Avantages du POM :**

* **Réutilisabilité accrue :** Les classes liées aux endpoints peuvent être partagées entre plusieurs scénarios de test.
* **Facilité de maintenance :** Une modification d’un endpoint ou d’un scénario n’impacte que la classe ou le fichier concerné.
* **Validation robuste :** La comparaison entre objets (POJOs) garantit une correspondance exacte entre les réponses des API et les attentes.

### **2.3.3 Tests Basés sur des Données (Data-Driven Testing)**

Les tests sont conçus pour être dynamiques et alimentés par des fichiers JSON.

**Principes de fonctionnement :**

* **Flexibilité :** Les cas de test peuvent être ajoutés ou modifiés via des fichiers JSON, sans nécessiter de changements dans le code source.
* **Paramétrisation par environnement :** Les fichiers JSON sont organisés en fonction des environnements (développement, test, production), assurant une configuration adaptée à chaque contexte.
* **Efficacité :** Une logique de test unique peut exécuter plusieurs scénarios grâce aux données paramétrées, augmentant la couverture des tests tout en réduisant les duplications.

## **2.4. Data-Driven Testing : Mise en Œuvre et Exemple Pratique**

Le Data-Driven Testing (DDT) implémenté dans le framework repose sur une architecture centrée sur la séparation des données et de la logique de test. Le fichier data-manager.json joue un rôle central dans cette approche, en organisant les données nécessaires pour chaque test sous forme d'array. Chaque élément de cet array correspond à un jeu de données spécifique pour un scénario de test particulier.

**Organisation des Données : Fichier data-manager.json**

Le fichier data-manager.json contient les données de test associées à chaque test case sous forme d'un tableau. Pour le test ValidateGetProfileTest, chaque index du tableau représente un jeu de données indépendant pour un scénario donné.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Figure 1 : Exemple de structure data-manager.json

Dans cet exemple :

* Le tableau contient deux jeux de données pour le test ValidateGetProfileTest avec son test ID ‘’Validate get profile’’.
* Chaque jeu est défini par des fichiers JSON (par ex., users.json, serverConfig.json, studentProfile.json) spécifiques, fournissant les données nécessaires à l'exécution.

**Chargement Dynamique des Données**

Lors de l'exécution, le framework récupère les données spécifiques pour le test en fonction de l'index du tableau correspondant au scénario en cours. Ce processus est entièrement automatisé et documenté dans les logs d'exécution**.**

Une image contenant capture d’écran, texte, Police

Description générée automatiquement

Figure 2 : Exemple de logs

Chaque fichier est chargé dynamiquement en fonction des instructions définies dans data-manager.json, assurant une flexibilité totale et une indépendance vis-à-vis du code source.

**Exécution et Résultats**

Dans cet exemple, le test ValidateGetProfileTest est exécuté deux fois, correspondant à deux scénarios distincts définis par les datasets (indices 0 et 1 du tableau dans data-manager.json). Cela garantit que la logique du test reste inchangée, tandis que seules les données varient.

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Figure 3 : Execution d'un suite de test

**Avantages de cette Approche**

1. **Flexibilité :**
   * Permet d'ajouter de nouveaux jeux de données en modifiant uniquement le fichier data-manager.json, sans toucher au code source.
2. **Réduction de la Duplication :**
   * Un même script de test peut couvrir plusieurs scénarios de données, réduisant ainsi la duplication du code.
3. **Maintenance Simplifiée :**
   * Les mises à jour ou corrections des données se font directement dans data-manager.json, garantissant une gestion centralisée et efficace.
4. **Efficacité et Reproductibilité :**
   * Chaque scénario est reproductible dans différents environnements grâce à la configuration dynamique des données.
5. **Transparence :**
   * Les logs détaillés montrent comment les données sont récupérées et utilisées pour chaque test, facilitant le suivi et le diagnostic.

Le Data-Driven Testing ainsi mis en œuvre dans le framework montre l'efficacité d'une approche orientée données, garantissant une couverture étendue des scénarios tout en minimisant les efforts de développement et de maintenance.

### **2.4.1. Gestion des Secrets**

Pour garantir la sécurité des données sensibles (clés API, tokens d’authentification), le framework intègre une gestion robuste des secrets.

**Approche sécurisée :**

* **Injection dynamique :** Les secrets sont récupérés depuis des gestionnaires sécurisés comme AWS Secrets Manager ou HashiCorp Vault et injectés sous forme de variables d’environnement lors de l’exécution.
* **Marquage des données sensibles** : Les valeurs dans les fichiers JSON sont marquées avec un préfixe spécifique ($en$itive), remplacées dynamiquement lors de l’exécution Centralisation sécurisée : Les secrets sont stockés de manière sécurisée en dehors du code source, minimisant les risques de fuites.

### **2.4.2. Génération de Rapports**

Le framework offre une visibilité complète et des analyses exploitables grâce à des rapports automatisés.

**Caractéristiques des rapports :**

* **Rapports dynamiques :** Fournissent des informations sur les tests réussis, échoués et ignorés, accompagnées de diagnostics détaillés**.**
* **Compatibilité CI/CD** : Les rapports sont formatés pour être intégrés dans des outils comme Jenkins ou GitHub Actions, permettant un suivi automatisé des résultats.
* **Analyse rapide :** Les rapports incluent des visualisations et des détails exploitables pour faciliter l’identification et la correction des anomalies.

Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement

Figure 4 : Rapport : Tous les tests réussis Capture d’écran affichant le résultat de l’exécution où tous les tests ont été validés avec succès.

Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Description générée automatiquement

Figure 5: Rapport : Échec d’un test avec détails Capture d’écran montrant un test échoué, avec une distinction claire des tests réussis, échoués, et ignorés.

Une image contenant texte, capture d’écran, Police, Page web

Description générée automatiquement

Figure 6 : Rapport : Différence entre les valeurs actuelles et attendues Capture d’écran détaillant l’échec d’un test, indiquant les logs, la valeur actuelle (actual value) et la valeur attendue (expected value).

## **2.5 Positionnement par rapport aux Outils Existants**

Le framework développé surpasse les solutions existantes grâce à plusieurs différenciateurs clés :

* **Modularité** : L’architecture basée sur le POM simplifie la maintenance et favorise la réutilisabilité.
* **Data-Driven Testing :** L’utilisation de fichiers JSON garantit une flexibilité accrue et une couverture étendue des tests.
* **Sécurisation avancée :** L’intégration de gestionnaires sécurisés pour les secrets élimine les risques de fuite des données sensibles.
* **Rapports détaillés :** Les résultats sont présentés de manière claire et structurée, compatibles avec les outils d’intégration continue.

Cette section a présenté l’architecture du framework développé, mettant en lumière ses principes de modularité, de réutilisabilité, et de compatibilité avec les environnements CI/CD modernes. Le design pattern Page Object Model (POM), l’approche data-driven, et la gestion centralisée des secrets positionnent ce framework comme une solution robuste, sécurisée et adaptée aux exigences des équipes de développement modernes.

# **CHAPITRE 3 : MÉTHODOLOGIE**

Ce chapitre détaille l’approche adoptée pour développer le framework d’automatisation des tests API. Il décrit la structure et les étapes clés de la mise en place, du développement et de la validation du framework, ainsi que l’organisation technique utilisée pour assurer sa modularité, sa maintenabilité, et son efficacité.

# **3.1 Approche Kanban**

L’approche Kanban a été choisie pour sa flexibilité et sa capacité à s’adapter à des flux de travail dynamiques. Elle permet de visualiser le processus de développement et de prioriser les tâches en fonction de leur importance immédiate.

**Organisation du Kanban :**

* **Colonnes utilisées :** 
  + ***Backlog* :** Liste des fonctionnalités et tâches à réaliser.
  + ***En cours* :** Tâches en cours de développement.
  + ***Tests*** : Tâches prêtes pour les tests de validation.
  + ***Terminé* :** Fonctionnalités finalisées et validées.
* **Priorisation des tâches :** Chaque tâche correspond à un objectif clair (ex. : ajout d’un support pour le data-driven testing, configuration de WireMock, gestion des secrets).
* **Suivi des progrès** : Des réunions hebdomadaires étaient organisées pour ajuster les priorités et résoudre les blocages.

Cette approche a permis une livraison progressive des composants, assurant une amélioration continue tout au long du développement.

## **3.2 Mise en Place**

### **3.2.1 Structure Maven**

La mise en place du framework a débuté par la création d’une structure Maven standard pour organiser le code de manière modulaire et évolutive.

* **Répertoire src/main/java :**

Contient les utilitaires, la logique métier et les composants principaux du framework (par exemple : clients HTTP, gestion des secrets, sérialisation JSON).

* + **Dossier utils :** Classes utilitaires pour les requêtes API, gestion des réponses, assertions et transformation des données.
  + **Dossier models :** Définit les objets de données utilisés pour représenter les réponses des API, basés sur le design pattern Page Object Model (POM).
* **Répertoire src/test/java :**

Contient les scripts de test organisés par modules et scénarios spécifiques (ex. : happy path, tests de régression).

* **Répertoire src/resources :**

Centralise les fichiers JSON pour les tests data-driven, les configurations spécifiques aux environnements (développement, test), et les logs.

**3.2.2 Intégration de WireMock**

WireMock a été configuré localement avec Docker Compose pour simuler des endpoints API. Cette approche garantit une isolation complète des tests et évite les dépendances à des services tiers.

**Configuration utilisée :**

* Un fichier docker-compose.yml a été défini pour exécuter WireMock sur le port 8081. Les stubs de simulation sont stockés dans un répertoire ./wiremock/mappings, facilement modifiable pour ajuster les réponses simulées.

Une image contenant capture d’écran, texte, logiciel, Logiciel multimédia

Description générée automatiquement

Figure 7 : Exemple de stub pour un endpoint simulé

Cette configuration garantit la reproductibilité des tests dans un environnement isolé.

## **3.3 Développement**

### **3.3.1 Préparation**

**Avant le développement, plusieurs étapes clés ont été réalisées :**

* **Analyse des besoins :** Identification des fonctionnalités nécessaires, telles que le data-driven testing, la gestion des secrets, et la génération de rapports.
* **Choix des outils :** Java a été choisi comme langage principal pour sa compatibilité avec des bibliothèques telles que RestAssured et TestNG.

### **3.3.2 Qualité du Code**

**La qualité du code a été maintenue grâce aux pratiques suivantes :**

* **Revues de code :** Chaque composant développé a été soumis à une revue pour s'assurer de sa lisibilité, de sa modularité, et de sa conformité aux standards de codage.
* **Conventions de codage :** Les conventions standard de Java ont été suivies, et des outils tels que Checkstyle et SonarLint ont été utilisés pour détecter les problèmes de qualité.

### **3.3.3 Architecture Modulaire**

**Le framework repose sur une architecture modulaire, facilitant sa maintenance et son évolutivité.**

1. **Design Pattern Page Object Model (POM) :**
   * Les endpoints API sont encapsulés dans des classes spécifiques, regroupant toutes les opérations associées (GET, POST, PUT, DELETE).
   * Les modèles POJO (Plain Old Java Object) représentent les réponses des API et permettent de comparer les résultats attendus (définis dans les fichiers JSON) avec les réponses obtenues.
2. **Data-Driven Testing :**
   * Les scénarios de test sont alimentés par des fichiers JSON, séparant les données des scripts de test pour une meilleure maintenabilité**.**
3. **Gestion des Secrets :**
   * Les secrets (clés API, jetons) sont injectés via des variables d’environnement au moment de l’exécution. Une classe dédiée transforme dynamiquement les données sensibles marquées dans les fichiers JSON (ex. : $en$itive:keyName).
4. **Génération de Rapports :**
   * Les résultats des tests sont exportés au format HTML et xml, avec des visualisations claires des tests réussis et échoués.

Ce chapitre a présenté une méthodologie structurée pour le développement d’un framework d’automatisation des tests API. L’approche Kanban, combinée à une architecture modulaire et à l’utilisation de WireMock, a permis de répondre efficacement aux besoins de validation des API. La qualité du code et l’organisation des tests basés sur des données garantissent une solution robuste et adaptable pour les environnements modernes**.**

# **CHAPITRE 4 : RÉSULTATS**

## **4.1 Fonctionnalités Testées**

Les fonctionnalités testées incluent une validation exhaustive des endpoints API, couvrant divers cas d'utilisation pour garantir leur conformité aux exigences fonctionnelles et non fonctionnelles.

**Liste des endpoints validés :**

* GET /api/v1/student/{id}/profile : Récupération des profils étudiants.
* PUT /api/v1/student/password : Mise à jour du mot de passe.
* PUT /api/v1/student/username : Mise à jour du nom d'utilisateur.

**Couverture atteinte :**

* **Nombre total de tests exécutés : 12**
* **Répartition des tests :**
  + Tests réussis : 12
  + Tests échoués : 0
  + Tests ignorés : 0
* **Temps total d'exécution** : 6.35 secondes

**Les tests incluent des validations spécifiques telles que :**

* La structure et les valeurs du corps JSON.
* Les statuts de réponse HTTP (200 pour les cas réussis).
* Les messages de réponse (succès ou échec d’une mise à jour).

## **4.2 Performances**

La mise en place de l'automatisation a permis une amélioration significative en termes de rapidité et de détection des anomalies :

* **Avant l’automatisation :**
  + Tests effectués manuellement, nécessitant un effort de configuration et d'exécution accru.
  + Temps moyen pour valider un endpoint : ~15 minutes.
  + Nombre moyen de tests par jour : ~10.
* **Après l’automatisation :**
  + Temps moyen pour valider un endpoint : ~0,1 seconde.
  + Nombre moyen de tests par jour : ~500.

**Améliorations constatées :**

* Réduction des délais d'exécution des tests de 95%.
* Détection anticipée des anomalies dans les environnements staging, réduisant les risques en production**.**

## **4.3 Tests Basés sur les Données (Data-Driven Testing)**

Le framework utilise un fichier data-manager.json pour centraliser et organiser les scénarios de test. Chaque jeu de données est associé à un ID de test et se charge dynamiquement avant l'exécution.

**Exemple :**Pour le test ValidateGetProfileTest, deux scénarios distincts ont été exécutés avec des jeux de données variés (indices 0 et 1 dans le fichier JSON). Le test reste identique tandis que les données changent.

**Avantages :**

* Gestion simplifiée des cas de test.
* Réutilisation accrue des scripts.
* Élimination des duplications dans le code.

## **4.4 Rapports Générés**

Les rapports fournis permettent une analyse claire et rapide des résultats des tests, offrant une vue d'ensemble complète des validations effectuées :

**Formats et outils :**

* Rapports HTML générés via TestNG.
* Visualisation des logs avec les détails des requêtes et des réponses (JSON).

**Exemples de rapports :**

1. **Rapport de tests réussis :**
   * Tous les scénarios validés avec succès.
   * Les détails incluent les statuts, les requêtes et les réponses.
2. **Rapport d’un test échoué :**
   * Mise en évidence des erreurs (valeurs actuelles vs attendues).
   * Logs détaillés pour diagnostic rapide.
3. **Rapport détaillé des scénarios :**
   * Inclut les requêtes envoyées (méthode, URI, headers, body).
   * Affiche les réponses reçues avec les codes d’état HTTP.

## **4.5 Exécution et Intégration CI/CD**

L'exécution des tests est simplifiée grâce à la commande Maven :

mvn clean test -Dgroups=all -DtestEnvironment=staging-ta

Cette méthode permet une intégration facile dans un pipeline CI/CD pour des validations automatiques à chaque mise à jour de code.

**Les résultats obtenus montrent l'efficacité du framework en termes de :**

* Validation complète des endpoints API.
* Réduction drastique des délais d’exécution des tests.
* Gestion des scénarios via une approche data-driven.
* Génération de rapports clairs et exploitables pour un suivi efficace.

# **CHAPITRE 5 : DISCUSSION**

## **5.1 Rétrospective**

Ce chapitre offre une rétrospective sur les résultats obtenus, en mettant en lumière les points positifs, les limitations rencontrées et leur impact global sur le projet.

### **5.1.1 Points Positifs**

Le projet a permis de créer un framework d’automatisation des tests API efficace et flexible, avec plusieurs avantages notables :

* **Productivité accrue :**
  + Réduction des efforts manuels grâce à l’automatisation des tests fonctionnels et de régression.
  + Simplification de l’ajout de nouveaux scénarios via le Data-Driven Testing.
* **Fiabilité des tests :**
  + Validation robuste des API en utilisant des données sérialisées dans des fichiers JSON, comparées directement avec les réponses des endpoints.
  + Simulation des dépendances externes à l’aide de WireMock, garantissant des tests isolés et reproductibles.
* **Organisation modulaire :**
  + Adoption du design pattern Page Object Model (POM), facilitant la maintenance et la réutilisabilité des composants.
* **Facilité d’exécution :**
  + Utilisation de commandes Maven (mvn clean test -Dgroups=all -DtestEnvironment=staging-ta) pour simplifier l’exécution locale et son intégration dans des pipelines futurs.

### **5.1.2 Points Négatifs**

**Malgré les succès, certaines limites ont été identifiées :**

* **Absence de gestion avancée des secrets :**
  + Faute de budget, des outils comme HashiCorp Vault ou AWS Secrets Manager n'ont pas pu être intégrés, ce qui a limité la sécurisation dynamique des données sensibles.
  + Les secrets ont été gérés localement via des variables d’environnement sans chiffrement avancé.
* **Tests non fonctionnels non pris en charge :**
  + Le framework ne couvre pas les tests de performance ou de sécurité, limitant l’évaluation complète des API.
* **Limitations en matière de collaboration :**
  + L'intégration avec Jira Xray pour gérer les plans de test et centraliser les résultats n’a pas été possible en raison des contraintes de licence.

## **5.2 Obstacles Rencontrés**

**Plusieurs défis ont été rencontrés au cours du projet, nécessitant des ajustements :**

* **Manque d’API ouvertes :**
  + La recherche d’API publiques adaptées aux tests s'est révélée infructueuse, ce qui a nécessité l’utilisation de WireMock pour simuler les endpoints à tester.
  + La configuration initiale de WireMock via Docker a demandé des efforts pour garantir la stabilité des environnements de test.
* **Limites de budget :**
  + L'absence de licence pour des outils comme AWS Secrets Manager et Jira Xray a contraint à développer des solutions locales ou à contourner certaines fonctionnalités avancées.
* **Complexité technique :**
  + La mise en place du Data-Driven Testing a requis une gestion minutieuse des fichiers JSON pour éviter des duplications ou des erreurs dans les scénarios.

## **5.3 Perspectives**

Pour renforcer le framework, plusieurs axes d'amélioration sont proposés :

### **5.3.1 Améliorations Techniques**

* **Ajout de tests non fonctionnels :**
  + Intégrer des outils comme JMeter pour les tests de charge et ZAP pour les tests de sécurité.
  + Étendre les scénarios pour inclure des tests de performance des endpoints.
* **Gestion avancée des secrets :**
  + Explorer des alternatives open source ou gratuites pour la gestion sécurisée des secrets, comme Vault by HashiCorp Community Edition.
  + Automatiser la rotation des secrets pour renforcer la sécurité.
* **Optimisation des simulations :**
  + Améliorer les configurations WireMock pour simuler des scénarios plus complexes et couvrir davantage de cas limites.

### **5.3.2 Améliorations Collaboratives**

* **Intégration avec des outils collaboratifs :**
  + Réévaluer l’utilisation de Jira Xray pour centraliser les résultats de test une fois une licence disponible.
  + Connecter les logs et rapports générés à des plateformes de suivi comme Confluence pour faciliter la documentation.
* **Documentation enrichie :**
  + Créer un guide utilisateur complet pour les développeurs et testeurs débutants, avec des exemples de mise en œuvre pour des tests fréquents.

### **5.3.3 Extensions Fonctionnelles**

* **Amélioration des rapports :**
  + Ajouter des visualisations interactives aux rapports pour faciliter l’analyse des anomalies.
  + Inclure des indicateurs de performance (temps de réponse, statut HTTP) directement dans les rapports.
* **Automatisation avancée :**
  + Développer des scripts supplémentaires pour l’automatisation complète des tests d’intégration, combinant plusieurs endpoints dans des workflows complexes**.**

Le projet a démontré son efficacité dans l’automatisation des tests API malgré des limitations techniques et budgétaires. Grâce à l’utilisation de WireMock, à une architecture modulaire basée sur le POM, et à une gestion structurée des données de test, le framework pose les bases d’une solution évolutive. Les améliorations futures envisagent une meilleure sécurisation, une couverture élargie avec des tests non fonctionnels, et une intégration renforcée dans des outils collaboratifs.

# **CHAPITRE 6 : CONCLUSION**

Ce projet marque une avancée significative dans l’automatisation des tests API REST en fournissant une solution modulaire, flexible, et adaptée aux besoins des environnements logiciels modernes. Le développement du framework a permis de surmonter plusieurs défis liés à la complexité des API, notamment grâce à l’utilisation de WireMock pour simuler des endpoints, à l’intégration du design pattern Page Object Model (POM) pour structurer le code, et à la mise en œuvre du Data-Driven Testing pour maximiser la couverture des scénarios.

Les résultats obtenus démontrent une amélioration substantielle de la productivité et de la fiabilité des tests, avec une réduction des délais d’exécution et une détection plus précoce des anomalies. La gestion des scénarios via des fichiers JSON a non seulement facilité la maintenance, mais aussi permis une adaptation rapide aux nouveaux cas d’usage.

Cependant, certaines limitations, telles que l’absence d’intégration avec des outils payants comme HashiCorp Vault ou Jira Xray, et le manque de tests non fonctionnels, montrent des axes d’amélioration pour le futur. Ces défis ouvrent des perspectives intéressantes pour renforcer la sécurité, étendre les capacités du framework, et intégrer des fonctionnalités collaboratives.

En conclusion, ce framework fournit une base solide et évolutive pour répondre aux besoins des équipes de développement et de test, tout en posant les jalons pour des extensions futures. Il s’agit d’un outil essentiel pour garantir la qualité et la stabilité des API dans un contexte où l’automatisation et la rapidité sont des priorités.

# **Bibliographie**

AbnAsia.org. “Comparing Different API Clients: Postman vs. Insomnia vs. ReadyAPI vs. Thunder Client vs. Hoppscotch.” *ABN Software*, 2 Apr. 2024, news.abnasia.org/blog/posts/en-comparing-different-api-clients-postman-vs-insomnia-vs-readyapi-vs-thunder-client-vs-hoppscotch-6729.

“Karate vs. Katalon Supports | Karate Labs.” *Karatelabs.io*, 2024, www.karatelabs.io/karate-vs-katalon.

“Karate vs. Postman | Karate Labs.” *Karatelabs.io*, 2023, www.karatelabs.io/karate-vs-postman.

Sharvan Bhatra. “Postman vs. Thunder Client: A Comparison of API Testing Tools and Alternatives.” *EchoAPI*, 28 Nov. 2024, www.echoapi.com/blog/postman-vs-thunder-client-a-comparison-of-api-testing-tools-and-alternatives/.