ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

RAPPORT TECHNIQUE
PRÉSENTÉ À L'ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
DANS LE CADRE DU COURS. GTI795 PROJET DE FIN D'ÉTUDES EN GÉNIE DES TI

# PFE-024 : GÉNÉRATION DE TESTS POUR SYSTÈME IOT

**PAR :**
MAXIMILIEN BLANCHARD-BIZIEN, BLAM13090804
ARMAND DIM DIM, DIM01048900

DÉPARTEMENT DE GÉNIE LOGICIEL ET DES TI

**Professeur-superviseur**

**Moha, Naouel**

MONTRÉAL, 12 AOÛT 2024
ÉTÉ 2024

**PFE024 : GÉNÉRATION DE TESTS POUR SYSTÈME IOT**

**Auteurs**
**MAXIMILIEN BLANCHARD-BIZIEN, BLAM13099804**
**ARMAND DIM DIM, DIM01048900**

## RÉSUMÉ

Le projet "Where is my Professor" (WIMP), développé par l'équipe Ptidej de l'Université Concordia, est une application Internet des objets (IoT) conçue pour améliorer la disponibilité des professeurs en temps réel pour les étudiants, en utilisant des dispositifs connectés tels que le robot "Buddy" et la montre "Fitbit". L'intégration de ces composants IoT pose des défis techniques en matière de tests, en raison de la diversité des protocoles de communication utilisés, comme WebSocket, MQTT, CoAP, et HTTP. La stabilité des connexions entre les composants est essentielle pour assurer la fiabilité des tests, car des erreurs de connexion peuvent entraîner des résultats incohérents, compromettant la validation du système (Tremblay & Gagnon, 2020).

Pour surmonter ces défis, un outil de gestion et d'exécution de tests a été développé, structuré autour d'une architecture de microservices. Cette approche permet de diviser le système en services indépendants, chacun ayant une fonction spécifique. Par exemple, le microservice d'authentification gère l'accès des utilisateurs via des jetons JWT, tandis que le microservice d'analyse des payloads valide les fichiers d'entrée avant de les transmettre au système WIMP qui gère les dispositifs IoT. Le microservice d'exécution des tests récupère et analyse les résultats des tests, en gérant les erreurs comme les timeouts. Enfin, un service registry centralise les communications entre les microservices, assurant une gestion efficace des requêtes et des performances optimisées.

Cet outil repose sur des technologies modernes telles que Node.js, Express.js, MongoDB, et des protocoles de communication IoT. L'interface utilisateur est développée avec Vue.js et Vuetify, offrant une gestion intuitive des tests. Le développement de l'outil a suivi une approche itérative avec une intégration continue (CI), permettant de valider chaque microservice de manière indépendante avant leur intégration dans l'ensemble du système. Des tests de performance ont été effectués pour vérifier la capacité de l'outil à traiter des payloads complexes sans délai significatif. Cependant, certaines limitations externes, telles que des problèmes de connexion, ont empêché la validation complète de certains cas d'utilisation, limitant ainsi l'atteinte de certains objectifs.

Cet outil améliore considérablement l'automatisation des tests IoT pour l'application WIMP, et son architecture flexible lui permet de s'adapter à d'autres systèmes IoT à l'avenir, renforçant ainsi la fiabilité et la qualité du système global.

# TABLE DES MATIÈRES

Page

**Introduction** **10**
**Chapitre 1: Vue d'ensemble du projet** **12**
1.1 Contexte 12
1.2 Problématique 12
1.3 Objectifs 13
**Chapitre 2: État de l'art** **14**
2.1 Méthodologie de Test IoT 14
2.1.1 Tests manuels 14
2.1.2 Tests automatisés 14
2.1.3 Méthodologie de test 15
2.2 Outils et Frameworks de test IoT 15
2.3 Défis et solution dans les tests IoT 16
2.4 Intérêt de l'outil de test dédié à WIMP 17
**Chapitre 3: Conception de l'outil de test pour WIMP** **19**
3.1 Architecture de l'outil 19
3.1.1 Microservice de l'authentification 20
3.1.2 Microservice d'analyse du payload 21
3.1.3 Microservice d'exécution des tests 21
3.1.4 Service Registry 22
3.1.5 L'interface utilisateur 23
3.2 Spécifications techniques et exigences 23
3.2.1 Spécifications fonctionnelles 23
3.2.2 Exigences non fonctionnelles 23
3.3 Modélisations 24
3.3.1 Diagramme de cas d'utilisation 24
3.3.2 Diagramme de séquence 25
3.3.3 Diagramme de paquetages 26
3.3.4 Diagramme de classe 26
3.3.4.1 Package AuthService 27
3.3.4.2 Package Payload 28
3.3.4.3 Package ExecuteTest 29
3.3.4.4 Package ServiceRegistry 30
**Chapitre 4: Implémentation et Validation** **31**
4.1 Environnement de développement 31
4.1.1 Configuration des outils 31

4.1.2 Gestion du versioning 31
4.2 Développement des microservices 31
4.2.1 Implémentation du microservice d’authentification 31
4.2.2 Implémentation du microservice d’analyse du payload 32
4.2.3 Implémentation du microservice d’exécution des tests 32
4.2.4 Configuration du service registry 33
4.2.5 Développement de l’interface utilisateur 33
4.3 Intégration et déploiement 33
4.3.1 Stratégie d'intégration continue (CI) 33
4.3.2 Stratégie de déploiement continu (CD) 33
4.4 Validation et vérification 34
4.4.1 Tests unitaires 34
4.4.2 Tests d'intégration 34
4.4.3 Tests de système 35
4.4.4 Exécution des tests et analyse des résultats 35
**Conclusion** **39**
**Bibliographies** **41**

# LISTE DES FIGURES


| | | Page |
|---|---|---|
| Figure 1 | Types de tests pour les systèmes IoT | 17 |
| Figure 2 | Diagramme de l'outil de test | 19 |
| Figure 3 | Diagramme du service d'authentification | 20 |
| Figure 4 | Diagramme de séquence pour l'analyse d'un payload | 25 |
| Figure 5 | Diagramme des paquetages des services | 26 |
| Figure 6 | Diagramme de classe du service "AuthService" | 27 |
| Figure 7 | Diagramme de classe du service "Payload" | 28 |
| Figure 8 | Diagramme de classe du service "ExecuteTest" | 29 |
| Figure 9 | Diagramme de classe du service "ServiceRegistry" | 30 |
| Figure 10 | Erreur durant la connexion car aucun courriel a été spécifié | 36 |
| Figure 11 | Analyse et envois du payload au WIMP | 37 |
| Figure 12 | Erreur de connexion avec l'application WIMP | 38 |
| Figure 13 | Résultats de tests affichés dans l'interface | 38 |

## LISTE DES ABRÉVIATIONS, SIGLES ET ACRONYMES

**WIMP** : Where is my Professor

**IoT** : Internet of Things

**MQTT** : Message Queuing Telemetry Transport

**CoAP** : Constrained Application Protocol

**HTTP** : Hypertext Transfer Protocol

**API** : Application Programming Interface

**Ptidej** : Nom de l'équipe de développement à l'Université Concordia

**JWT** : JSON Web Token

**CI** : Continuous Integration (Intégration Continue)

**CD** : Continuous Deployment (Déploiement Continu)

# Introduction

Les systèmes "Internet des objets" (IoT) sont très complexes, mais offrent des possibilités considérables pour étendre les fonctionnalités d'une application. Par exemple, un outil de surveillance de la santé peut envoyer des informations à une application, qui à son tour peut transmettre des instructions à un autre logiciel. Cependant, tester et automatiser l'ensemble du système devient un défi de taille. Bien qu'il soit possible de tester individuellement chaque composant du système en utilisant des outils comme JUnit, Selenium, etc., ces approches ne permettent pas de tester efficacement les interactions entre les différents composants. Tester l'ensemble du système IoT en tant qu'unité fonctionnelle est particulièrement difficile et sujet à des erreurs, compromettant la fiabilité et la performance globale.

L'application "Where is my Professor" (WIMP), développée par l'équipe Ptidej de l'Université Concordia, vise à résoudre le problème de la disponibilité des professeurs pour les étudiants. En utilisant différents outils tels qu'un petit robot nommé "Buddy" et une montre de santé appelée "Fitbit", cette application permet de déterminer si les professeurs sont à leur bureau et disponibles. Cela aide les étudiants à savoir quand il est opportun de visiter leurs enseignants pour poser des questions. La fiabilité de cette application repose sur une intégration harmonieuse et stable de ses composants IoT, qui doivent interagir sans faille pour fournir des informations en temps réel.

L'application WIMP fonctionne et offre des services de base, mais elle présente des défis inhérents à sa structure IoT. Chaque composant individuel peut être testé de manière isolée, mais il est essentiel de garantir que WIMP maintienne une connexion stable et efficace avec tous les outils et composants du système. En cas de défaillance de la connexion, les tests risquent de générer des erreurs non souhaitées, compromettant ainsi leur fiabilité. Ces erreurs de connexion imprévues peuvent conduire à des résultats de tests inconsistants, compliquant la validation et l'assurance qualité de l'application WIMP.

L'objectif de notre projet est de développer un outil de gestion et d'exécution de tests pour l'application "Where is my Professor" (WIMP) de l'équipe Ptidej. Cet outil analysera les fichiers d'entrée appelés payloads et enverra les commandes appropriées à WIMP pour être exécutées sur les appareils ciblés. Le système retournera un message de succès ou d'échec en fonction du résultat du test, incluant des messages de timeout en cas de problèmes de connexion.

L'outil sera conçu avec une architecture de microservices pour faciliter les modifications et la correction de code. Une première version fonctionnelle est attendue après deux mois, suivie de mises à jour continues jusqu'à la fin du projet. Le succès de l'outil sera mesuré par sa capacité à traiter correctement au moins 95% des fichiers d'entrée et à fournir des rapports détaillés pour chaque cas de test. En outre, l'outil devra être capable de s'adapter aux nouveaux changements et mises à jour du système, assurant ainsi une couverture de test exhaustive et une performance optimale de l'application WIMP.

# Chapitre 1: Vue d'ensemble du projet

## 1.1 Contexte

Dans les établissements scolaires, comme les cégeps ou les universités, pouvoir poser des questions directement aux professeurs est important pour un étudiant. Cependant, il peut être difficile pour eux de pouvoir savoir quand ces professeurs sont disponibles. Oui, ils ont un horaire, mais il est possible qu'ils ne les poursuivent pas à la lettre, probablement dû à un événement imprévu dans leur vie, par exemple.

L'application "Where is my Professor", ou tout simplement "WIMP", compte régler ce problème. Créé par l'équipe Ptidej de Concordia, ce logiciel utilise différents outils, comme un petit robot appelé "Buddy" et une montre de santé portant le nom de "Fitbit", pour déterminer si les professeurs des établissements sont à leur bureau ou non, et donc, s'ils sont disponibles ou non. Grâce à ça, les étudiants pourront savoir quand ce serait un bon moment pour aller visiter leurs enseignants ou non, s'ils ont besoin de leur poser des questions.

## 1.2 Problématique

L'application WIMP fonctionne et offre des services de base, mais elle présente néanmoins des défis inhérents à sa structure basée sur l'Internet des objets (IoT). Parmi ces défis, l'écriture de tests s'avère particulièrement complexe. En effet, rédiger une suite de tests pour chaque composant individuel de l'application ne pose pas de difficulté significative. Cependant, tester l'ensemble du système devient nettement plus ardu.

Les membres de l'équipe Ptidej doivent garantir que WIMP maintienne une connexion stable et efficace avec les autres outils et composants du système. Cette exigence est essentielle pour assurer la fiabilité et la performance de l'application. Toutefois, en cas de défaillance de la connexion, les tests risquent de générer des erreurs non souhaitées, compromettant ainsi leur fiabilité et leur cohérence. Ces erreurs de connexion imprévues peuvent conduire à des résultats de tests inconsistants, compliquant davantage la validation et l'assurance qualité de l'application WIMP. En résumé, bien

que la fonctionnalité individuelle de WIMP soit assurée, la complexité de son écosystème IoT pose des défis significatifs pour l'écriture et la gestion des tests à l'échelle du système entier.

## 1.3 Objectifs

L'objectif de notre projet est de développer un outil de gestion et d'exécution de tests pour l'application "Where is my Professor" (WIMP) de l'équipe Ptidej. Cet outil aura les fonctionnalités suivantes :

- **Analyse des Fichiers d'Entrée (Payloads) :**

L'outil analysera les payloads fournis en entrée pour vérifier leur contenu. Si le contenu du payload est correct, il enverra les commandes appropriées à l'application WIMP.

- **Envoi et Exécution des Commandes :**

Les commandes seront exécutées sur les appareils ciblés par l'application WIMP. Le système surveillera l'exécution des commandes pour en garantir le bon déroulement.

- **Retour des Résultats :**

Le système retourne un message de succès ou d'échec en fonction du résultat du test. En cas de problèmes de connexion, des messages de timeout seront retournés.

- **Architecture de Microservices :**

L'outil sera conçu avec une architecture de microservices pour faciliter les modifications et la correction de code.Cette architecture permettra une maintenance et des mises à jour aisées de l'outil.

- **Planning et Livrables :**

Une première version fonctionnelle de l'outil est attendue après trois mois de développement et les mises à jour continues seront effectuées jusqu'à la fin du projet pour améliorer et affiner l'outil.

- **Critères de Réussite :**

Le succès de l'outil sera mesuré par sa capacité à traiter correctement au moins 95% des fichiers d'entrée. L'outil devra fournir des rapports détaillés pour chaque cas de test, incluant des informations sur les succès, les échecs et les timeouts.

En résumé, cet outil vise à automatiser et à améliorer le processus de test pour l'application WIMP, en assurant une gestion efficace des tests et une analyse précise des résultats.

# Chapitre 2: État de l’art

L’Internet des objets, ou “IoT” est une structure ou processus qui consiste à connecter différents objets à l'Internet, et les relier dans un réseau. Ceci permet à ces objets de communiquer et envoyer des informations entre eux, mais peut causer des difficultés pour tester le système en entier, surtout parce qu’il faut maintenir une connexion stable entre tous les objets.

Le test des systèmes IoT (Internet of Things) est un domaine en pleine expansion, en raison de la croissance rapide des technologies connectées. Les systèmes IoT intègrent divers dispositifs, réseaux, services infonuagiques et applications, ce qui rend leurs tests complexes et multidimensionnels. Cet état de l'art explore les méthodes actuelles, les outils et les défis associés au test des systèmes IoT, en se concentrant sur les approches manuelles et automatisées, ainsi que sur les méthodologies de test courantes.

## 2.1 Méthodologie de Test IoT

### 2.1.1 Tests manuels

Les tests manuels des systèmes IoT impliquent des interventions humaines pour vérifier la fonctionnalité, l'interopérabilité et la performance des dispositifs et des systèmes. Ces tests sont essentiels pour identifier les problèmes liés à l'expérience utilisateur et aux interactions réelles entre les dispositifs. Cependant, ils sont souvent chronophages, sujets à des erreurs humaines et difficiles à reproduire de manière cohérente.
Mais selon **Yalantis**[1], les tests fonctionnels manuels restent essentiels pour les projets IoT et sont applicables aux premières étapes de développement ainsi qu'à travers l'affinement du produit. Cependant, en raison de la complexité croissante des écosystèmes IoT, l'automatisation devient une nécessité pour la plupart des solutions.

### 2.1.2 Tests automatisés

Les tests automatisés utilisent des scripts et des outils logiciels pour exécuter des tests prédéfinis sur des systèmes IoT. Ils sont particulièrement efficaces pour les tests de régression, permettant de vérifier rapidement si de nouvelles modifications ont introduit des bugs dans le système. Les outils

[1] https://yalantis.com/blog/iot-testing-guide/

d'automatisation peuvent également exécuter des tests de performance à grande échelle, simulant des milliers d'interactions simultanées pour évaluer la robustesse du système. Néanmoins, l'automatisation complète des tests IoT reste un défi en raison de la diversité des dispositifs et des technologies impliquées.

### 2.1.3 Méthodologie de test

Les principales méthodologies de test pour les systèmes IoT comprennent :

- **Black Box Testing (test boîte noire)** : Cette approche teste le système sans connaissance interne de son fonctionnement. Les tests sont basés sur les spécifications et les exigences du système, vérifiant si les sorties correspondent aux entrées attendues;
- **White Box Testing (test boîte blanche)** : Cette approche utilise une connaissance complète du code source et de la structure interne du système. Les tests visent à vérifier la logique interne, les chemins de code et la couverture des tests;
- **Gray Box Testing (test boîte grise)** : Cette approche combine des éléments des tests boîte noire et tests boîte blanche. Les testeurs ont une connaissance partielle du système, ce qui leur permet de concevoir des tests plus ciblés et efficaces.

## 2.2 Outils et Frameworks de test IoT

- **JUnit :**

JUnit est un framework de test unitaire populaire pour Java, largement utilisé pour tester les applications Android dans les systèmes IoT. Il permet aux développeurs d'écrire et d'exécuter des tests unitaires, facilitant la détection précoce des bugs et des problèmes de code.

- **Mocha :**

C’est un framework de test JavaScript utilisé pour les applications Node.js. Il fournit une structure pour organiser et exécuter des tests, avec des fonctionnalités telles que les hooks before/after pour configurer et nettoyer les tests, ainsi que des assertions pour vérifier les résultats des tests.

- **WebDriverIO :**

C’est un outil d'automatisation de navigateur basé sur WebDriver, utilisé pour les tests de bout en bout des interfaces utilisateur. Il permet de simuler des interactions utilisateurs complexes avec des applications web, garantissant que l'interface fonctionne correctement sur différents navigateurs et plateformes.

Dans le domaine des systèmes IoT (Internet of Things), les tests sont cruciaux pour garantir la qualité, la sécurité et la performance des dispositifs interconnectés. Bien qu'il existe déjà divers outils de test, chaque outil a ses propres limites et peut ne pas répondre entièrement aux besoins spécifiques de tous les projets IoT. Le test des systèmes IoT étant un domaine complexe et en évolution rapide, nécessitant une combinaison de tests manuels et automatisés pour garantir la qualité et la fiabilité des systèmes. Les méthodologies de test doivent s'adapter à la diversité des dispositifs et des technologies, tandis que les outils d'automatisation et les avancées récentes, comme l'IA et les environnements virtuels, offrent ainsi de nouvelles opportunités pour améliorer l'efficacité et la couverture des tests.

Le projet de développement d'un outil de test pour l'application WIMP s'inscrit dans cette dynamique, visant à fournir une solution robuste et flexible pour gérer et exécuter des tests spécifiques dans un environnement IoT complexe.

## 2.3 Défis et solution dans les tests IoT

La diversité des dispositifs et des protocoles pose un défi majeur. Par exemple, tester un système IoT qui intègre des capteurs de différentes marques nécessite des scénarios de test variés pour garantir l'interopérabilité.

- **Hétérogénéité des Systèmes IoT** :

Les systèmes IoT intègrent une variété de dispositifs, chacun utilisant des technologies, des protocoles de communication et des langages de programmation différents. Les outils existants ne peuvent pas tous supporter cette diversité de manière homogène. L’outil de test personnalisé peut être conçu pour répondre spécifiquement aux besoins du projet "Where is my Professor" (WIMP), intégrant parfaitement les particularités des dispositifs, des réseaux, et des services cloud utilisés.

- **Intégration et Interopérabilité** :

Les dispositifs IoT doivent souvent interagir avec des systèmes tiers. Tester l'interopérabilité entre ces systèmes nécessite des scénarios de test complexes. L’outil de test peut être développé pour gérer ces scénarios d'interopérabilité, assurant que tous les composants du système WIMP fonctionnent correctement ensemble.

- **Gestion des Erreurs et des Exceptions** :

Les systèmes IoT sont souvent vulnérables aux erreurs de connexion et aux défaillances de réseau. Les outils existants peuvent ne pas gérer efficacement toutes les exceptions spécifiques aux environnements IoT. L'outil de test peut être conçu pour détecter, gérer et rapporter de manière efficace les erreurs spécifiques aux systèmes IoT, comme les timeouts et les interruptions de connexion.

- **Automatisation Personnalisée** :

Bien que les outils d'automatisation existants, comme JUnit, Mocha, et WebDriverIO, soient puissants, ils peuvent nécessiter une configuration et une intégration spécifiques pour fonctionner efficacement avec le système WIMP. L'outil de test personnalisé peut automatiser non seulement les tests unitaires et de bout en bout, mais aussi les tests de performance et de sécurité, adaptés spécifiquement aux besoins du projet.

Les systèmes IoT doivent pouvoir évoluer pour gérer un grand nombre de dispositifs. Par exemple, tester la scalabilité d'un réseau de capteurs environnementaux implique de simuler l'ajout de nouveaux capteurs et de mesurer l'impact sur le réseau.

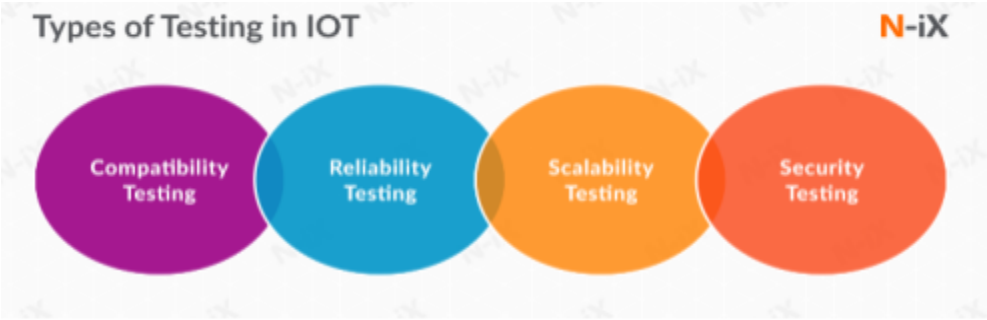


**Figure 1 : Types de test IoT**

## 2.4 Intérêt de  l'outil de test dédié à WIMP

Chaque outil de test ayant ses propres limites et ne pouvant répondre entièrement aux besoins spécifiques de tous les projets IoT comme celui avec WIMP, la création d'un nouvel outil de test pour ce projet présente des avantages significatifs. Répondre spécifiquement aux besoins uniques du projet, d'optimiser les processus de test, de réduire les coûts et les délais, et d'améliorer la qualité et la fiabilité du système IoT. En développant un outil de test personnalisé, l'équipe Ptidej peut s'assurer

que le système WIMP fonctionne de manière optimale dans un environnement IoT complexe et en constante évolution.

Pour donner quelques exemples, certains outils donnés, comme “QA Wolf”, sont des services payants, ce qui ne serait pas très pratique pour Concordia et l’équipe Ptidej, qui devront payer des frais mensuels pour continuer d’utiliser ces services. D’autres, comme “Bevywise IoT Simulator” ou “Wireshark”, sont plus des outils de simulation ou de suivi que des outils pour envoyer et exécuter des cas de tests, les rendant un mauvais choix pour Ptidej. En effet, les cas de tests doivent être automatisés, ce que ces outils ne peuvent pas faire, car ils ne font que suivre comment se déroule le système IoT. Il serait donc mieux de créer notre propre outil de tests, qui dit a WIMP ce qu'il doit faire, et leur envoyer un message selon s’il a été capable de le faire ou non, plutôt que d’utiliser un outil existant.

Pour des exemples plus spécifiques, nous avons plusieurs choix qui ne marchent pas correctement. Par exemple, “JUnit” aurait l’air d’être un bon choix, mais vu que l’outil est mieux utilisé pour des applications orientées objets, c’est un mauvais choix pour WIMP, qui est une application IoT. Pour d’autres outils de tests, comme “Mocha”, le problème vient plus du fait que l’équipe Ptidej veut spécifiquement tester le système au complet, et pas juste le code dans chacun des outils utilisés, donc, “Mocha” ne sera pas utile. Les outils de tests de boîte noire comme “Selenium” ou “Cypress” ne s’appliqueront pas non plus, car WIMP n’a pas une interface assez développée pour pouvoir interagir avec. Le meilleur choix courant est “Playwright”, mais il a aussi des problèmes. Spécifiquement, vu que WIMP manque encore certaines fonctionnalités, “Playwright” aura donc des difficultés à interagir avec l’application en utilisant son interface sur un navigateur, surtout pour envoyer les requêtes aux autres outils dans le système.

De plus, créer notre propre outil peut permettre un plus grand contrôle sur ce qui peut être testé. Par exemple, si les membres de l’équipe Ptidej veulent utiliser l’outil sur d’autres applications IoT, et pas juste sur WIMP, ils pourront tout simplement modifier notre application pour les prendre en compte.

# Chapitre 3: Conception de l'outil de test pour WIMP

La conception de l'outil de test pour l'application "Where is my Professor" (WIMP) nécessite une approche méthodique pour s'assurer qu'il répond aux exigences spécifiques du projet. Cet outil doit être capable d'analyser des fichiers d'entrée, d'envoyer des commandes à WIMP pour les exécuter sur les appareils ciblés, et de retourner des messages de succès ou d'échec en fonction des résultats des tests. Ce chapitre détaille les différentes phases de la conception de cet outil, en mettant l'accent sur l'architecture, les composants principaux, et les technologies utilisées.

## 3.1 Architecture de l'outil

L'outil de test sera basé sur une architecture de microservices. Cette architecture permet de diviser le système en services indépendants, chacun responsable d'une partie spécifique du processus de test. Les microservices communiquent entre eux via des APIs RESTful, ce qui facilite l'extensibilité et la maintenance de l'outil. Les principaux composants de l'architecture incluent :

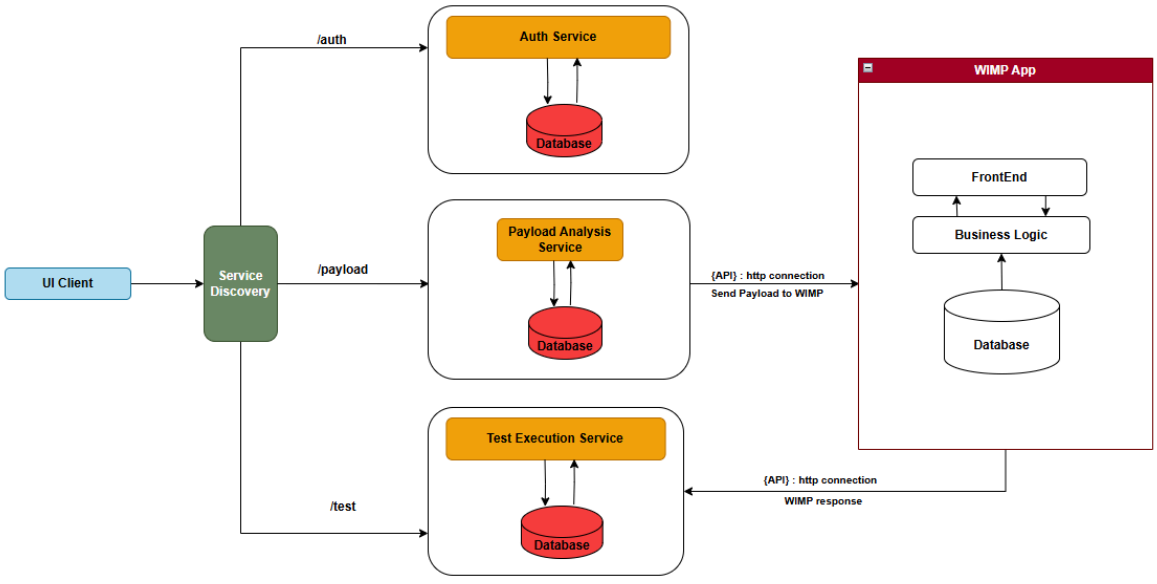


**Figure 2 : Diagramme de l'outil de test**

- Microservice d'authentification;
- Microservice d'analyse du payload;

- Microservice d'exécution des tests;
- Service registry;
- Interface utilisateur pour la gestion et la supervision des tests.

### 3.1.1 Microservice de l'authentification

Ce microservice est responsable du système d'authentification pour les utilisateurs. En particulier, il confirme que les informations données par l'utilisateur pour se connecter sont correctes, et ajoute un nouvel utilisateur, en utilisant les informations données dans un autre formulaire. De plus, c'est ce service qui génère un jeton pour l'utilisateur lorsque celui-ci s'est connecté avec succès, et pour qu'il reste connecté. Les technologies utilisées sont les suivantes :

- **Express.js :** framework pour créer des APIs RESTful;
- **JWT-Token:** pour la génération et la vérification des jetons d'authentification;
- **MongoDB:** base de données pour stocker les informations des utilisateurs;
    - **Bcrypt:** utilisé pour hacher les mots de passe des utilisateurs.

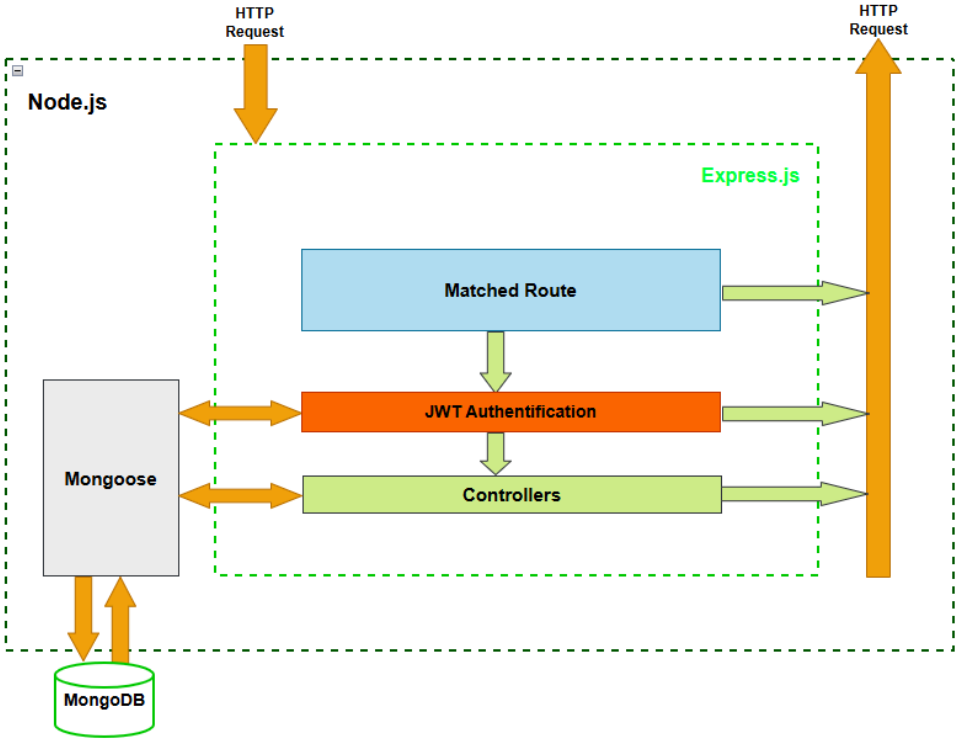


**Figure 3 : Diagramme du service d'authentification**

### 3.1.2 Microservice d’analyse du payload

Ce composant est responsable de la réception et de la validation des fichiers d'entrée. Il doit s'assurer que le format et le contenu du  fichier est correct avant de l’envoyer et créer les connexions vers l’application IoT ciblé. Ce service s’occupe aussi d’une moitié des connexions utilisées pour communiquer avec l’application cible. Spécifiquement, il s’occupe de la partie qui envoie le contenu du payload vers WIMP, s’assurant d’abord que la connexion soit bel et bien établie avant de l’envoyer.

Grâce à son architecture flexible, il peut être facilement étendu pour inclure des fonctionnalités supplémentaires ou pour s'adapter à des exigences ou changements spécifiques. La combinaison de Node.js, Express et MongoDB assure une performance et une scalabilité élevées, tandis que l'utilisation d'Axios, de WebSockets et/ou d’autres protocoles (MTTQ, CoAP, HTTP)  permet une communication efficace avec des services externes et internes. En résumé, nous avons utilisées comme technologie :

- **WebSocket:** utilisé pour les communications en temps réel et bidirectionnel, particulièrement utile dans les applications nécessitant une faible latence et une communication continue entre le client et le serveur;
- **MQTT:** utilisé principalement dans les systèmes IoT pour communiquer efficacement entre des dispositifs et des capteurs ayant des contraintes de bande passante et de puissance;
- **CoAP:** utilisé dans les environnements IoT similaires à MQTT, mais avec une approche RESTful pour faciliter l'interaction avec les systèmes web existants;
- **HTTP:** utilisé de manière générale pour la communication web, les API RESTful, et les services web, avec un accent sur l'interopérabilité et la flexibilité;
- **Node.js:** pour le serveur backend;
- **Express.js:** pour la création de l'API RESTful
- **Axios:** pour envoyer les requêtes HTTP vers les autres services.

### 3.1.3 Microservice d’exécution des tests

Une fois que le service d’analyse du payload a envoyé le payload, ce service attend que l’application WIMP retourne une bonne partie des résultats de tests. Ensuite, avec une requête de l’utilisateur, il ira chercher les résultats des tests exécutés, et sauvegardera les nouveaux résultats dans sa base de données. Après ça, l’application retournera tous les résultats sauvegardés, et qui sont reliés à un

certain identifiant donné. Ce composant doit également gérer les éventuels timeouts. Ce microservice où est géré les résultats (succès ou échec) de test permet de générer des rapports détaillés. Il doit aussi analyser les résultats et fournir des rapports compréhensibles pour les développeurs et les testeurs. Ce microservice peut également inclure des fonctionnalités de notification pour alerter l'équipe en cas d'erreurs critiques et les technologies utilisées ici sont :

- **Node.js:** pour le serveur backend;
- **Express.js:** pour la création de l'API RESTful;
- **MongoDB:** pour stocker les résultats des tests;
- **Axios:** pour envoyer les requêtes HTTP vers les autres services;
- **WebSocket:** pour la communication et le reçus de réponses en temps réel;
- **MQTT:** utilisé principalement dans les systèmes IoT pour communiquer efficacement entre des dispositifs et des capteurs ayant des contraintes de bande passante et de puissance;
- **CoAP:** utilisé dans les environnements IoT similaires à MQTT, mais avec une approche RESTful pour faciliter l'interaction avec les systèmes web existants;
- **HTTP:** utilisé de manière générale pour la communication web, les APIs RESTfuls, et les services web, avec un accent sur l'interopérabilité et la flexibilité.

### 3.1.4 Service Registry

Pour qu'une application microservice fonctionne correctement, il faut utiliser soit un "Service Registry", soit un "Gateway", un outil qui permet de gérer quelles requêtes doivent être envoyées dans quels services. Voici le but de ce microservice, qui sert principalement à unir les différents microservices du backend dans un seul lien "localhost". Par exemple, en utilisant le "localhost" du "Service Registry", on pourra non seulement appeler les requêtes du microservice d'exécution des tests, mais aussi de l'analyse du payload de la même manière. Aussi, ce service s'occupe de s'assurer que les requêtes soient répondues de façon rapide, en incluant une limite de temps que les requêtes doivent prendre avant de retourner une erreur de timeout, et une limite de requête qui peut être envoyé, pour éviter les problèmes de latences. Ce microservice utilise les technologies suivantes:

- **Node.js**: pour le serveur backend;
- **Express.js**: pour la création de l'API RESTful;
- **http-proxy-middleware:** pour la création et centralisation des appels vers les autres microservices.

### 3.1.5 L’interface utilisateur

Cette interface contient tous les éléments graphiques de l’application. Elle est principalement codée avec Vue.js et Vuetify pour les composants, et utilise aussi le gateway pour pouvoir facilement appeler et envoyer des requêtes aux autres services. L’interface utilise principalement une implémentation à une page, où les étapes pour envoyer un payload et obtenir les résultats de test sont sur une seule page. L’interface utilise les technologies suivantes :

- **Vue.js:** framework JavaScript pour construire l'interface utilisateur.
- **Vuetify.js:** bibliothèque de composants Vue.js pour un design matériel.
- **Axios:** pour les requêtes HTTP vers les APIs backend.

## 3.2 Spécifications techniques et exigences

### 3.2.1 Spécifications fonctionnelles

Les spécifications fonctionnelles définissent les diverses fonctionnalités que l'outil doit offrir pour répondre aux besoins des utilisateurs. L'authentification des utilisateurs est un aspect crucial, incluant la gestion des comptes et la génération de jetons JWT pour assurer la sécurité et l'authenticité des identités. La validation et l'analyse du payload sont également couvertes, détaillant le processus de vérification et de validation des fichiers d'entrée avant leur transmission pour les tests. L'exécution et la gestion des tests sont des composantes essentielles, décrivant les procédures mises en place pour réaliser les tests et gérer leurs résultats. La gestion des résultats de test est abordée, expliquant comment les résultats sont stockés, analysés et présentés de manière claire et compréhensible aux utilisateurs. Enfin, des exigences spécifiques sont établies pour l'interface utilisateur intuitive, afin de garantir une utilisation facile et efficace de l'outil.

### 3.2.2 Exigences non fonctionnelles

Elles comprennent les performances et la scalabilité, définissant les attentes en matière de rapidité, de temps de réponse et de capacité à gérer une charge de travail croissante. La sécurité et la confidentialité sont également des aspects primordiaux, présentant les mesures de protection des données et de préservation de la confidentialité des informations. La fiabilité et la disponibilité sont des critères essentiels pour garantir un service continu et sans interruption, assurant ainsi la satisfaction des utilisateurs. Enfin, la maintenabilité et l'extensibilité sont des éléments clés pour

faciliter la maintenance de l'outil et permettre son extension future, en ajoutant de nouvelles fonctionnalités pour répondre aux besoins évolutifs des utilisateurs.

## 3.3 Modélisations

### 3.3.1 Diagramme de cas d'utilisation

Le diagramme de cas d'utilisation montre les principales interactions entre les utilisateurs (utilisateurs et administrateurs) et les microservices du système.

**Microservice AuthService:**

- **S'inscrire:** l'utilisateur s'inscrit en fournissant les informations nécessaires;
- **Se connecter:** l'utilisateur s'authentifie pour accéder à l'outil de test.

**PayloadService:**

- **Charger un payload:** l'utilisateur charge un fichier d'entrée pour les tests;
- **Valider le payload:** le microservice valide le format et le contenu du payload;
- **Envoyer un payload:** l'utilisateur envoie le payload validé au WIMP pour exécution;
- **Sauvegarder un payload:** sauvegarde l'historique du payload dans la base de donnée;
- **Obtenir l'historique des payloads:** récupérer l'historique des payloads.

**ExecuteTestService:**

- **Ajouter un résultat de test:** ajoute un nouveau résultat de test dans la base de données;
- **Obtenir résultats de test:** récupère les résultats des tests pour une date spécifique;
- **Obtenir tous les résultats de test:** récupère tous les résultats des tests;
- **Sauvegarder les nouveaux résultats de test:** le microservice récupère les résultats de tests de WIMP qui ont un certain identifiant, et sauvegarde tous les nouveaux résultats de tests qui ne sont pas déjà dans la base de données.

**ServiceRegistry:**

- **Créer des proxy:** crée les proxy pour les services définis;
- **Gérer les services:** gère les services enregistrés et la redirection des requêtes;
- **Limiter le taux de requêtes:** limite le taux de requêtes et définit un timeout pour chaque requête.

### 3.3.2 Diagramme de séquence

Le diagramme de séquence pour l'envoie d'un payload dans le système "Where is my Professor" (WIMP) montre les interactions entre les différents composants du système lorsqu'un utilisateur charge un payload, et comment ce payload est validé et sont ensuite sauvegardés dans une base de données MongoDB.

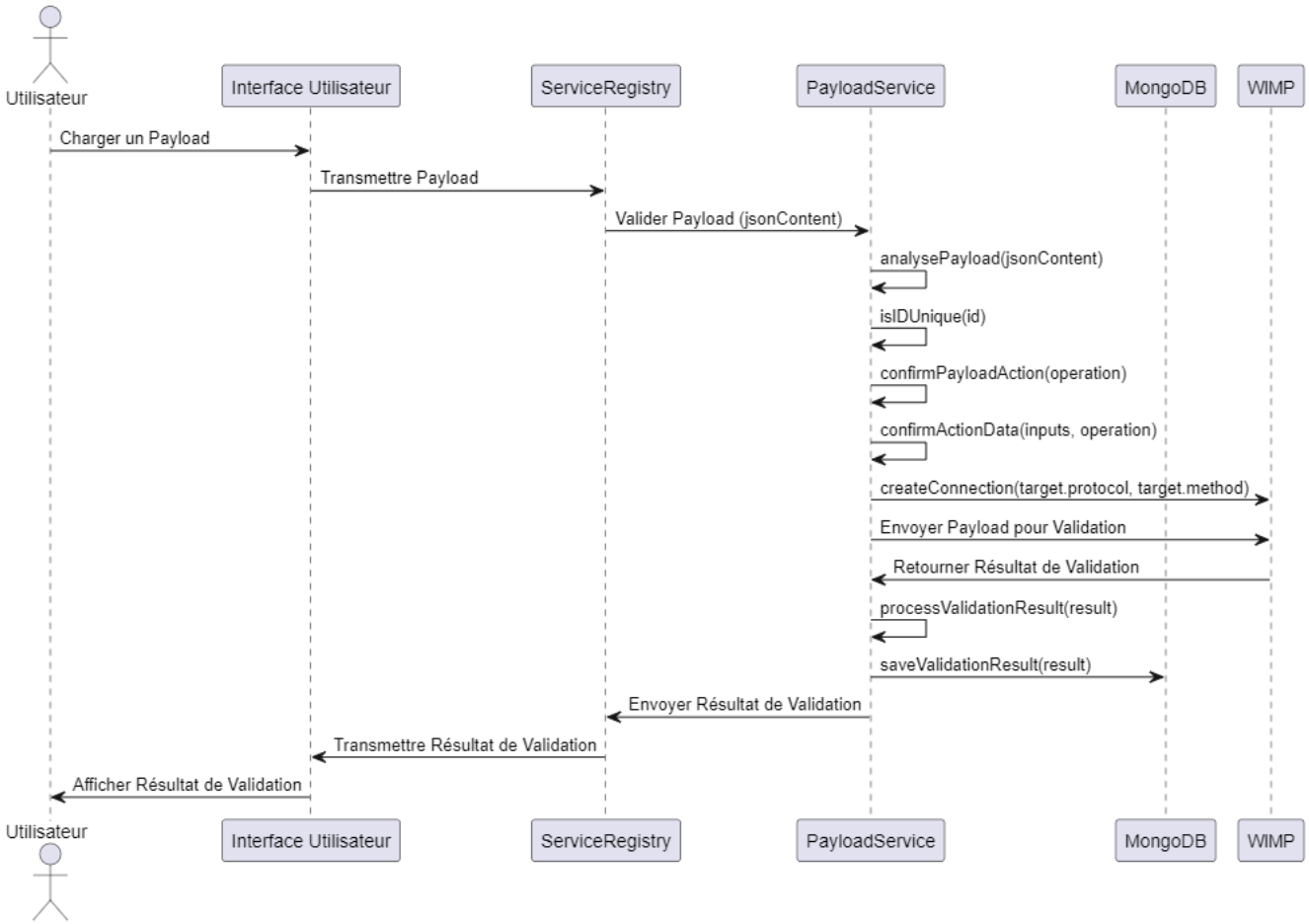


**Figure 4: Diagramme de séquence pour l'analyse d'un payload**

### 3.3.3 Diagramme de paquetages

Un diagramme de paquetages en UML est utilisé pour représenter la structure organisationnelle des éléments d'un système en regroupant des classes, des interfaces, des composants, et d'autres éléments similaires en paquets logiques. Il aide à visualiser la modularité du système et les dépendances entre les différents composants. Un paquet est donc une représentation d'un conteneur regroupant des éléments du modèle (classes, interfaces, etc.).

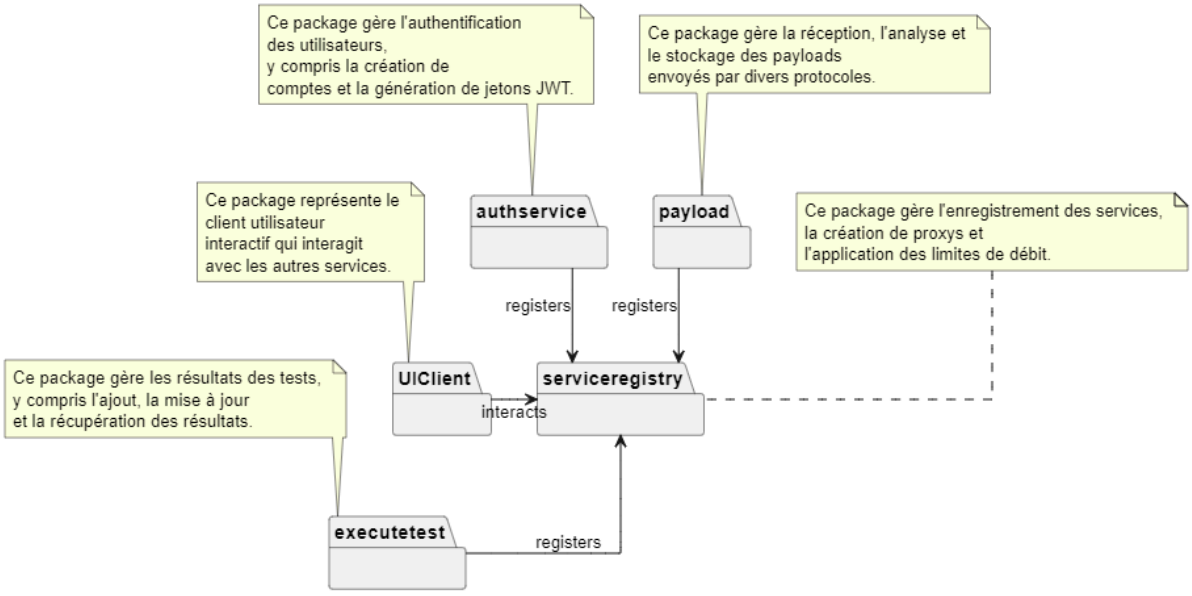


**Figure 5: Diagramme des paquetages des services**

### 3.3.4 Diagramme de classe

Les diagrammes de classe suivants montrent les relations et les interactions entre les différents microservices du système de test, y compris leurs classes principales, attributs, méthodes et les liens entre eux. Étant donné que les relations sont déjà présentes dans le diagramme de paquetages ci-dessus, nous allons représenter chaque paquet avec ses différentes classes.

### 3.3.4.1 Package AuthService

Ce paquet gère l'authentification des utilisateurs, y compris la création de comptes et la génération de jetons JWT.

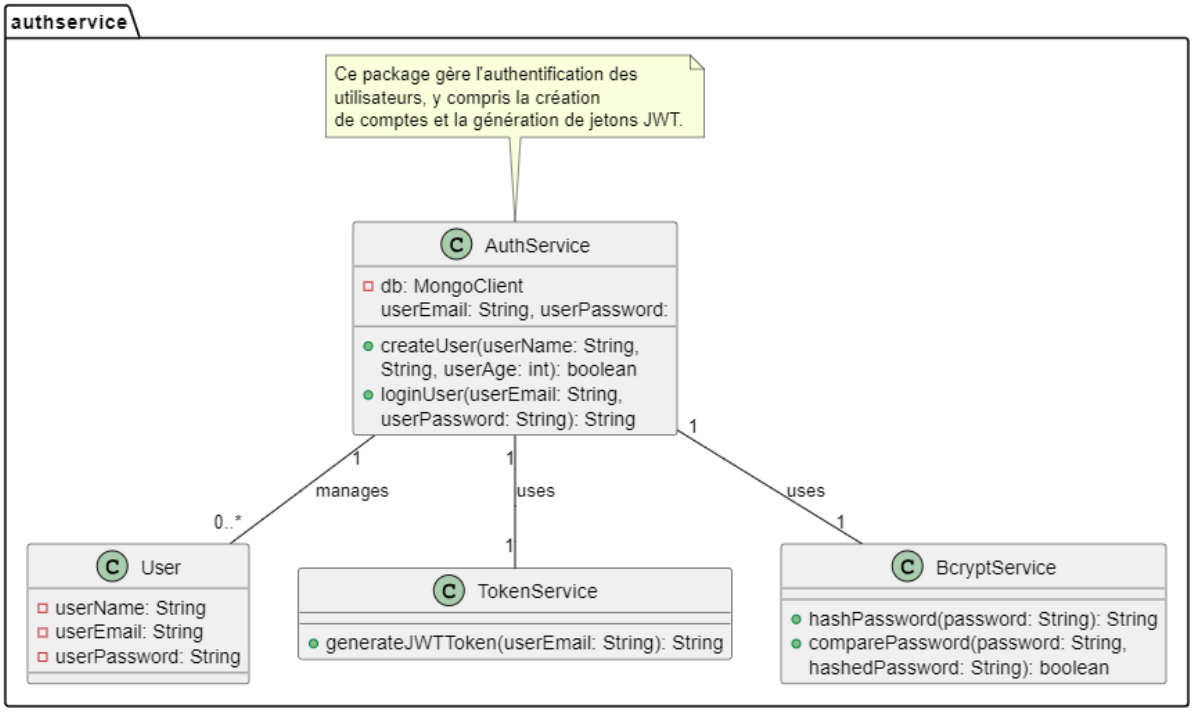


**Figure 6: Diagramme de classe du service “AuthService”**

### 3.3.4.2 Package Payload

Ce paquet gère la réception, l'analyse et le stockage des payloads envoyés par divers protocoles comme “WebSocket”, “CoAP”, “HTTP” et “MQTT”. Il reçoit les données, valide leur structure et contenu, puis les stocke de manière sécurisée avec des métadonnées associées. Le paquet maintient un historique détaillé des payloads pour des suivis futurs et assure une gestion efficace des connexions avec des modules spécifiques pour chaque protocole. Il permet ainsi une intégration fluide et une gestion complète des données reçues.

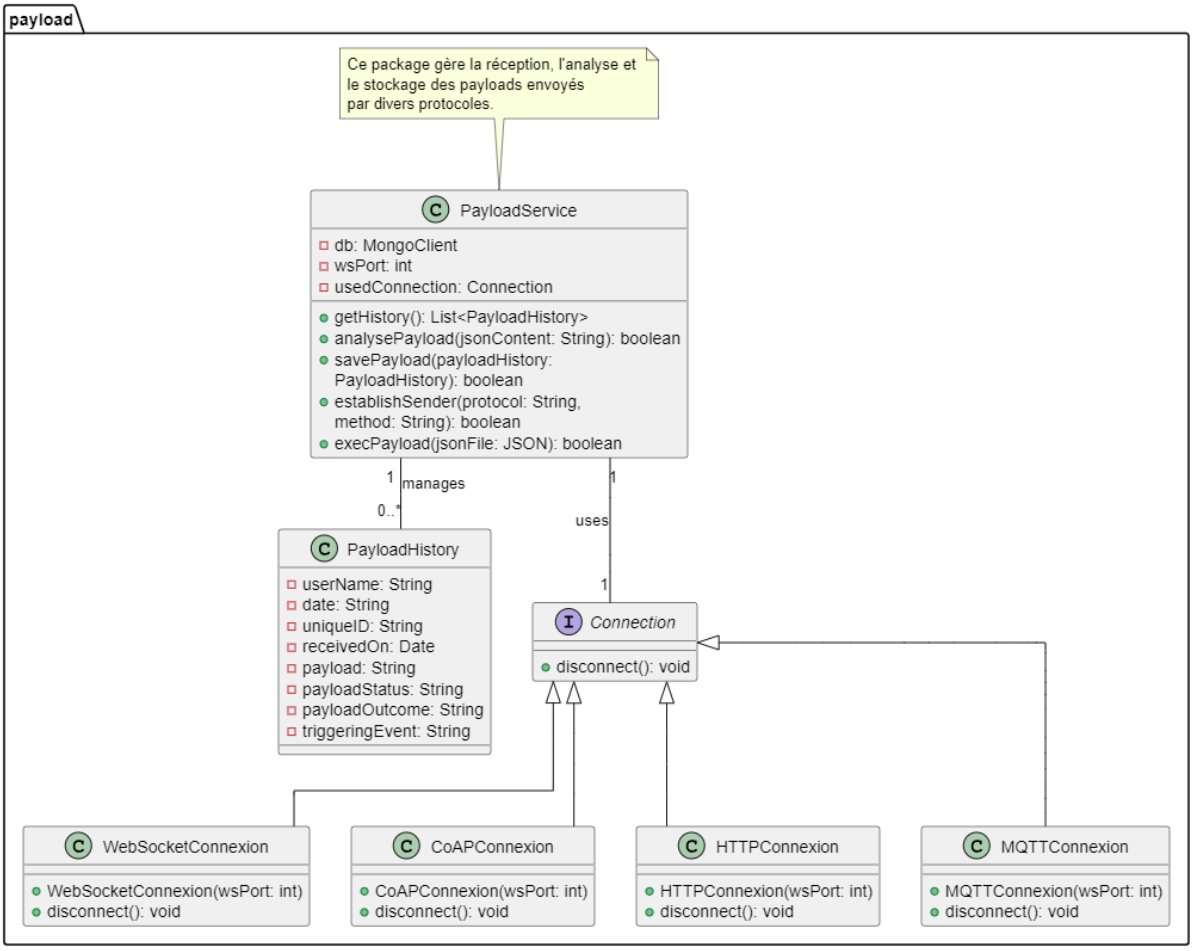


**Figure 7: Diagramme de classe du service “Payload”**

### 3.3.4.3 Package ExecuteTest

Le paquet “ExecuteTest” gère les résultats des tests, couvrant l'ajout, la mise à jour et la récupération des résultats. Il permet de stocker les résultats des tests exécutés, d'actualiser les résultats existants en cas de réexécution des tests, et de récupérer les résultats pour des analyses futures.

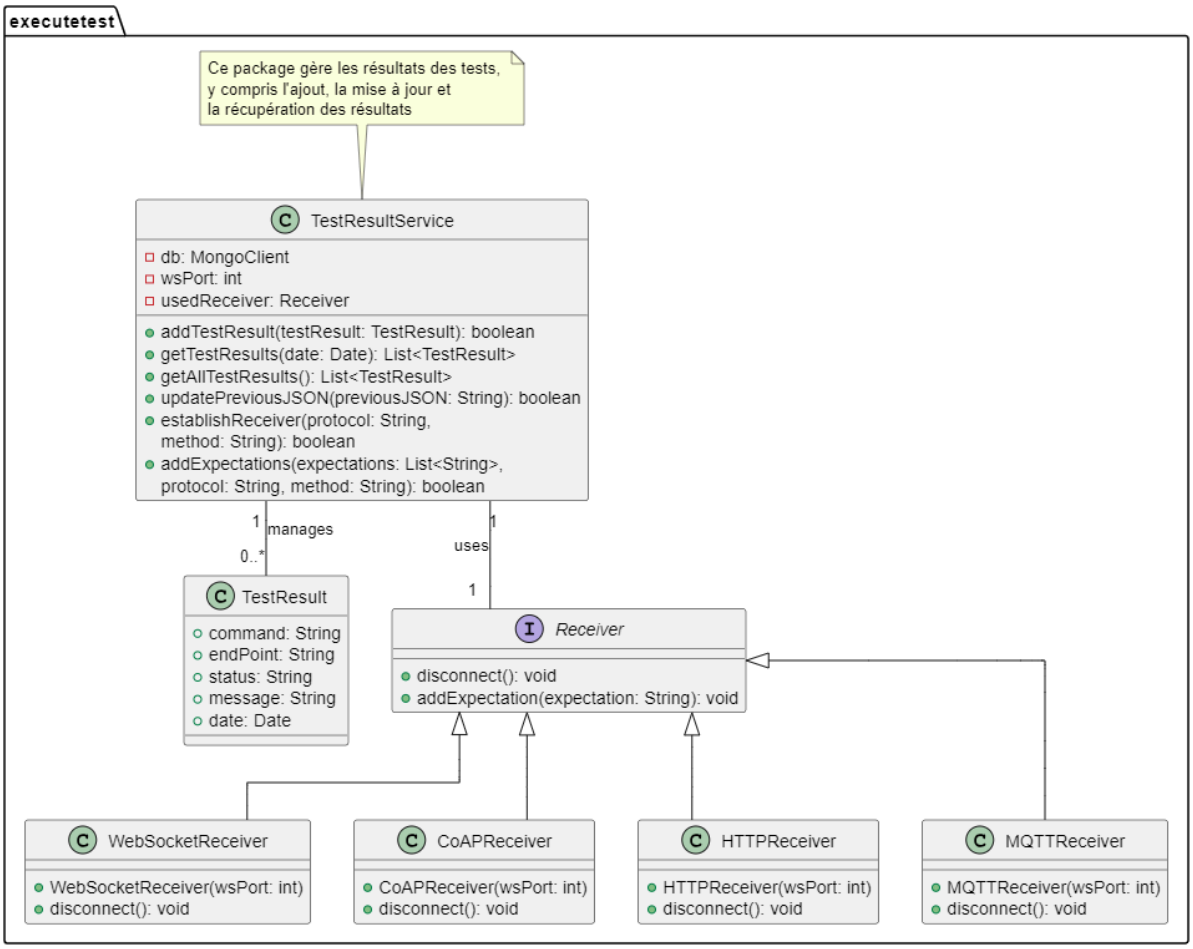


**Figure 8: Diagramme de classe du service “ExecuteTest”**

### 3.3.4.4 Package ServiceRegistry

Ce paquet gère l'enregistrement des services, la création de proxys et l'application des limites de débit. Il maintient une liste de services disponibles, crée des proxys pour rediriger les requêtes et utilise un module de limitation de débit pour contrôler le trafic réseau. En surveillant les requêtes et en appliquant des règles de limitation, ce package assure la disponibilité et la performance des services. Il facilite également l'intégration et la communication sécurisée entre les différents composants du système.

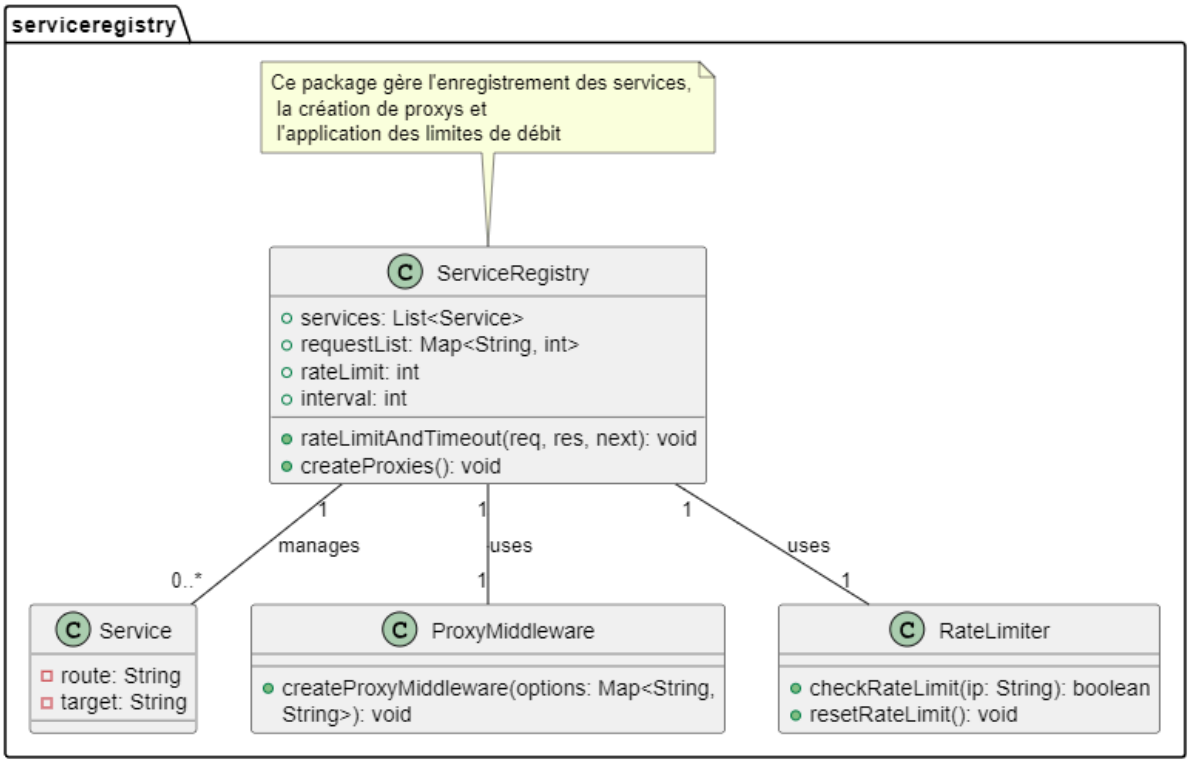


**Figure 9: Diagramme de classe du service “ServiceRegistry”**

# Chapitre 4: Implémentation et Validation

## 4.1 Environnement de développement

### 4.1.1 Configuration des outils

Le backend représente la majorité des services dans notre application. Pour rouler toutes nos applications en même temps, incluant aussi le frontend, nous avons utilisé l'outil "Concurrently", qui permet de rouler plusieurs scriptes de façon parallèle, et sur un seul terminal dans Visual Studio Code. Tout ceci est fait dans le "programme" qui se trouve à la base du dossier de projet, et ne fait qu'exécuter les scripts qu'on spécifie avec la commande "npm run".

### 4.1.2 Gestion du versioning

Les versions de notre application sont basées sur les changements qu'on lui apporte. Chaque fois qu'on pousse des changements sur notre branche principale, une nouvelle version de l'application est créée. Ceci nous permet de facilement retourner à une version précédente si de nouveaux changements brisent notre application, ou si les changements apportés ne sont pas satisfaisants. Aussi, si on a une branche qui prend du temps à fusionner, au point où la branche principale avance et obtient d'autres fonctionnalités pendant ce temps, on peut juste utiliser "GitHub Desktop" pour mettre à jour cette branche à partir de la branche principale.

## 4.2 Développement des microservices

### 4.2.1 Implémentation du microservice d'authentification

Pour l'authentification, nous avons principalement créé plusieurs événements Express qui s'occupent de gérer la connexion et la création des utilisateurs. En particulier, pour la connexion, on fait un appel à un événement qui regarde dans la base de données des utilisateurs, et tente de retourner un résultat qui est pareil que le nom d'utilisateur et le mot de passe donné. Ensuite, la requête retourne un jeton JWT si un utilisateur est trouvé, ou un message d'échec si aucun élément n'est trouvé ou l'un des deux paramètres demandés n'a pas été donné. L'autre événement principal consiste à créer un nouvel utilisateur. Après avoir confirmé que tous les éléments sont là, et que le nom d'utilisateur et le courriel sont uniques, on crée un nouvel utilisateur dans la base de données.

À part ça, le microservice s'occupe aussi de générer un jeton JWT pour maintenir et confirmer la connexion d'un utilisateur valide. Pour la création, nous utilisons le nom de l'utilisateur, et une

confirmation disant s'il est un administrateur ou non, selon si son nom contient le mot "admin", et un secret qui est une série de chiffres et caractères généré aléatoirement. Ce jeton dure une heure, et utilise l'algorithme "HS256".

### 4.2.2 Implémentation du microservice d'analyse du payload

Pour l'analyse du payload, nous avons mis en place une méthode dynamique pour initier la connexion, où on prend les premières informations de connexion, et on crée la connexion selon le protocole spécifié. Par exemple, si le payload dit qu'il faudra utiliser une connexion WebSocket, l'application créera une instance des classes "WebSocketConnexion". Les quatres classes de connexion courantes sont très similaires, mais traitent différents types de connexions, et s'occupent principalement d'envoyer le payload spécifié.

Aussi, comme pour les autres microservices, l'analyse du payload s'occupe de gérer sa propre base de données. Dans ce cas, cette base de données contient les informations des payloads qui ont été analysées et qui sont valides, incluant la date qu'ils ont été soumis. Le microservice peut ajouter de nouveaux payloads valides, obtenir tous les payloads sauvegardées, et rechercher les payloads qui ont été soumis autour d'une certaine période de temps. Il y a aussi deux événements pour effacer les éléments dans la base de données: un pour enlever toutes les entrées, et un autre pour enlever toutes les entrées qui ont un certain identifiant appelé "TC_ID".

### 4.2.3 Implémentation du microservice d'exécution des tests

Comme pour le microservice de l'analyse du payload, nous avons mis en place une méthode dynamique pour initier les connexions requises. La différence principale est que les connexions dans ce service ne font qu'attendre les résultats retournés par l'application IoT. Toutes ces classes utilisent une méthode dans le fichier index du service qui ajoute un nouveau résultat de test dans la base de données. De plus, ces classes ont une instance d'une connexion HTTP vers un lien qui demande a WIMP d'aller chercher les tests exécutés par l'application, basés sur l'identifiant du payload appelé "TC_ID", et les sauvegarder dans la base de données de MongoDB sous un format plus simple.

À part ça, le fichier index du service contient les événements qui s'occupent de gérer les éléments dans la base de données, tel qu'ajouter un résultat de test, obtenir les résultats de tests d'une certaine date ou identifiant de payload, ou "TC_ID", ou de tout simplement chercher tous les résultats de tests disponibles. Aussi, comme pour le service d'analyse du payload, ce service peut aussi effacer

rapidement les éléments de sa base de données. Il peut soit effacer tous les résultats sauvegardés, ou soit effacer tous les résultats qui ont un certain “TC_ID”.

### 4.2.4 Configuration du service registry

Pour le service registry, nous avons utilisé “http-proxy-middleware” pour établir le service. En utilisant cet outil, nous avons établi une liste des routes qui seront écrites sur le port local utilisé, et la véritable route que le registry va appeler quand cette route est référencée. Donc, par exemple, si on appelle “localhost:3006/payload”, le service registry va appeler “localhost:3002/payload”, en passant les données dans le corps en référence. Ensuite, pour chaque route, on associe des données pour que l’application puisse les appeler avec succès. Aussi, pour empêcher que les requêtes se bloquent infiniment, nous avons aussi mis en place une méthode qui met en place une limite de temps, et une limite de requête. Si la requête ne retourne pas une réponse dans le temps alloué, le programme qui a envoyé la requête retournera une erreur de “timeout”. D’une même façon, si trop de requêtes sont envoyées au service registry dans un certain laps de temps, alors la requête retournera une erreur disant que trop de requêtes ont été envoyées.

### 4.2.5 Développement de l’interface utilisateur

## 4.3 Intégration et déploiement

### 4.3.1 Stratégie d'intégration continue (CI)

Pour éviter des problèmes de conflits lorsque nous travaillons ensemble sur le projet, nous avons tout simplement utilisé la fonction des branches sur Github. Lorsque nous devons modifier ou ajouter une certaine fonctionnalité, nous créons une nouvelle branche, et travaillons dessus. Ensuite, lorsque nous sommes sûrs que cette tâche spécifique est complétée, on ouvre un “Pull Request”, et on fait ensuite un “merge”. Pour les changements mineurs, comme corriger rapidement certaines erreurs d’orthographe, on ne fait qu’appliquer ces changements dans la branche principale parce qu’ils ne devraient pas causer des problèmes de conflits.

### 4.3.2 Stratégie de déploiement continu (CD)

Vu que notre code utilise une architecture microservice, nos stratégies de développements sont très dynamiques et adaptatives. En particulier, si un problème arrive, ou si un changement est requis dans l’un des services, nous pouvons toujours interagir avec les autres services sans problèmes. En particulier, vu que la majorité des méthodes et fonctions sont des événements Express, nous pouvons

tout simplement tester ces fonctionnalités en faisant un appel direct à l'élément en utilisant un outil comme Postman, et en entrant manuellement les données dans le corps de la requête. Ceci nous permet d'intégrer de nouveaux changements rapidement, de les tester sans avoir besoin de rouler tous les services et, si des problèmes arrivent, de ne pas avoir à arrêter tout le code de notre logiciel.

## 4.4 Validation et vérification

### 4.4.1 Tests unitaires

Tester individuellement chaque service ne devrait pas être un si grand problème. Vu que notre programme utilise une architecture microservice, nous pouvons tout simplement tester chaque service individuellement. Les seules exceptions devraient être les fonctions ou services qui demandent l'utilisation d'un autre service, comme le "Service Registry", ou les appels que le service du payloads doit faire vers le service d'exécution des tests. De plus, nous pouvons utiliser "Postman" pour facilement et rapidement envoyer des requêtes au service qu'on est en train de tester.

Donc, en général, lorsque nous aurons implémenté une nouvelle fonction sur le code du backend, d'abord, nous exécuterons le service qui a reçu cette fonction, et tous autres microservices si c'est nécessaire pour le fonctionnement de la fonction. Ensuite, nous utiliserons Postman pour envoyer une requête à cette nouvelle implémentation, en mettant dans le corps de la requête tous les paramètres nécessaires.

### 4.4.2 Tests d'intégration

Nous avons commencé avec un test simple pour tester l'analyse des payloads, qui demande l'interaction de deux services en particulier pour pouvoir fonctionner. Spécifiquement, l'une des requêtes dans le fichier JSON consiste à demander au robot Buddy d'avancer, et à une certaine vitesse. Cependant, la vitesse spécifié pour le déplacement est trop grande, ce qui causera l'analyse de retourner une erreur disant que la vitesse doit être entre un certain nombre.

Ensuite, nous avons fait un test avec une variation du dernier payload, qui devrait marcher correctement. En particulier, le fichier demandera a WIMP d'aller chercher le battement de cœur à partir de la montre FitBit. Ensuite, il déterminera l'action que le robot buddy doit prendre en dépendance du battement du cœur. Dans ce cas-ci, l'application devra retourner un message disant de déplacer le robot. Finalement, WIMP demandera au robot de se déplacer un peu vers l'avant, et ce

dernier doit retourner un message disant que le mouvement a été effectué avec succès. Après que WIMP ait fini tous les tests, on enverra une requête pour obtenir les résultats disponibles, et on les sauvegardera dans la base de données, en disant si ils ont été réussis ou non en comparant le message attendu, et le message qui a été vraiment retourné.

Pour notre troisième test avec les intéractions avec le backend, nous allons faire un test simple où nous allons demander à WIMP d'activer les roues du robot Buddy, et de renvoyer un message. La raison de ce test est de confirmer que notre code peut confirmer qu'un cas de test peut être réussi, en spécifiant la réponse que WIMP retournera dans le message attendu.

Pour notre quatrième cas de test majeur, le cas consiste tout simplement à envoyer un payload qui contient une requête pour exécuter toutes les actions que notre analyseur de payload peut couramment prendre en compte. Avec ce test, nous pourrons non seulement mieux voir les actions et requêtes que WIMP peut traîter, mais aussi voir les messages qu'il peut retourner ou non, et à quoi s'attendre pour les tests à venir.

Finalement, notre dernier test sera un test de performance pour l'analyse du payload. Spécifiquement, les membres de Concordia nous ont dit que les fichiers payloads qu'ils vont générer auront entre 100 et 200 opérations à faire. Donc, nous allons analyser un payload qui contient environ 150 opérations, pour voir si notre algorithme peut analyser le payload sans prendre trop de temps.

### 4.4.3 Tests de système

Pour les tests avec tous les services de notre application, nous allons principalement tester les évènements, intéractions, etc. entre le frontend et le backend, en utilisant l'interface que l'utilisateur va interagir avec. Comme ça, nous pourrons nous assurer que l'application fonctionne correctement dans sa somme, et qu'il n'y aura pas de problèmes quand l'utilisateur utilisera l'application. Aussi, si des problèmes arrivent, nous pourrons facilement les voir dans la console du mode inspection.

### 4.4.4 Exécution des tests et analyse des résultats

Pour les résultats des tests unitaires, ils ont été réussi avec succès. En général, si une requête retourne une erreur, on peut facilement trouver où l'erreur a été lancée, surtout si on gère les messages d'erreurs nous-mêmes. D'une même façon, pour les fonctions qui demandent d'envoyer une requête à

un autre service, les tests ont aussi fonctionné avec succès. Les services peuvent tout de même fonctionner ensemble, même si tous les services ne sont pas activés.

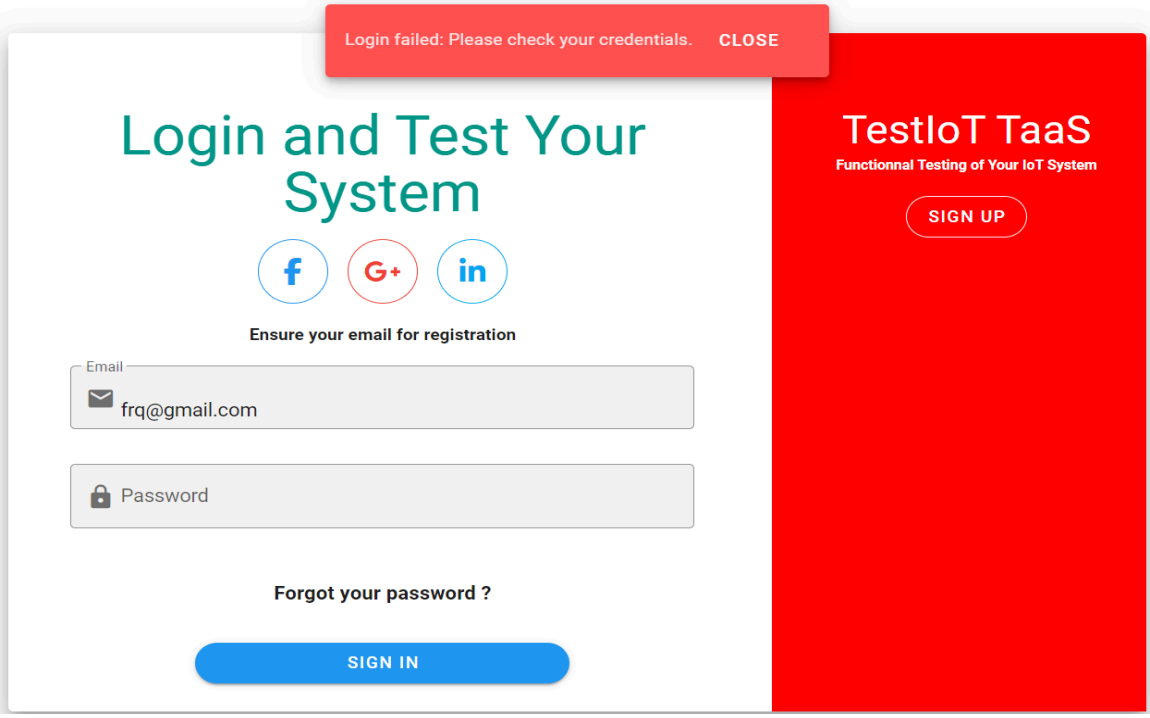


**Figure 10: Erreur durant la connexion car aucun courriel a été spécifié**

Lors des tests d'intégration avec les fichiers payloads, plusieurs problèmes ont été rencontrés. Les deux premiers fichiers ont bien fonctionné, mais l'un d'entre eux est devenu obsolète. Plus précisément, le premier cas, qui consistait à analyser un fichier contenant des données incorrectes et à renvoyer une erreur indiquant une valeur incorrecte, n'est plus pertinent. En effet, le code a récemment été modifié pour ne plus vérifier si une valeur dépasse une certaine limite, mais seulement pour s'assurer que la valeur est présente et du bon type. Après discussion avec les membres de l'équipe Ptidej, il a été convenu qu'il serait préférable que ce soient les outils et le système IoT eux-mêmes qui gèrent ces erreurs ou alertes, plutôt qu'une application externe.

On peut donc voir l'analyse et  l'envoi d'un payload réussir avec l'exécution de plusieurs opérations sur le robot Buddy

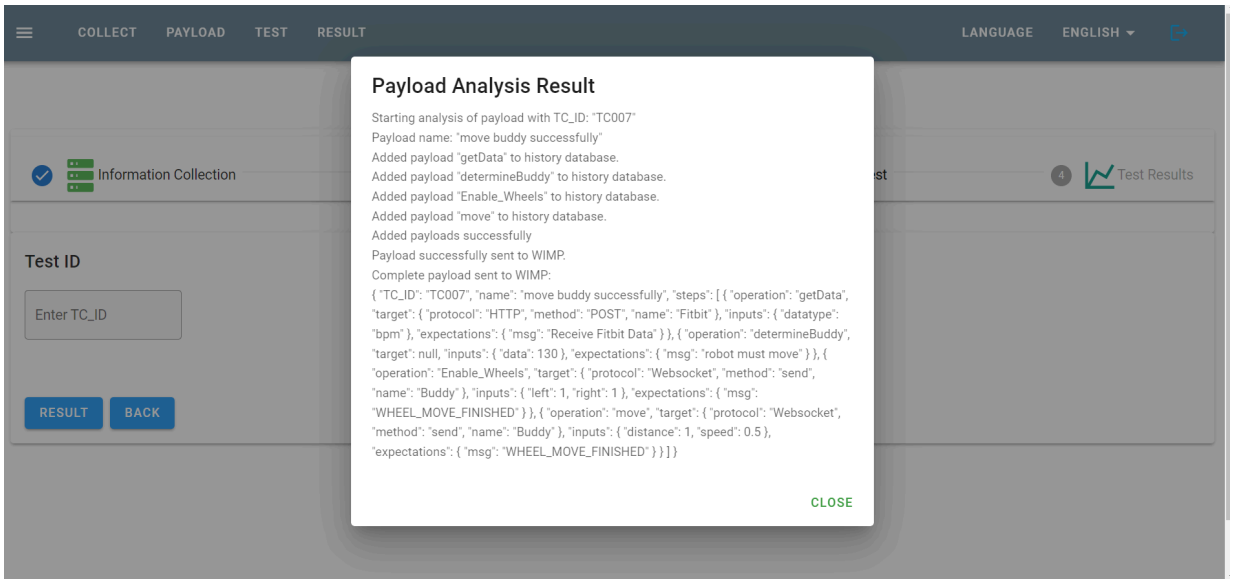


**Figure 11: Analyse et envois du payload au WIMP**

Cependant les deux autres cas de tests, qui impliquent l'envoi de payloads réussis et l'exécution de plusieurs opérations sur le robot Buddy, n'ont pas pu être réalisés en raison d'un problème indépendant de notre volonté. En particulier, la connexion que nous utilisions pour envoyer des requêtes à WIMP a soudainement cessé de fonctionner. Bien que nous puissions toujours communiquer avec WIMP et envoyer des requêtes, il n'était plus possible de sauvegarder les résultats en raison d'un manque d'accès à Internet, ce qui empêchait l'accès à nos bases de données MongoDB. Malgré ce contretemps, le test de performance impliquant l'analyse de 150 opérations différentes a été un succès. L'algorithme a rapidement terminé l'analyse de ces opérations et les a sauvegardées dans la base de données. Cela nous permet d'affirmer avec confiance que notre application est capable de gérer la majorité des payloads, quelle que soit leur taille. En résumé, sur les quatre cas de tests envoyés à WIMP, deux ont été complétés, un est devenu obsolète, et les deux autres n'ont pas pu être réalisés en raison de problèmes de connexion à Internet. Cependant, le test de performance a été réussi sans latence ni retard.

```
[3] Added payloads successfully.
[3] node:events:495
[3]       throw er; // Unhandled 'error' event
[3]       ^
[3]
[3] Error: connect ETIMEDOUT 10.42.0.75:8484
[3]     at TCPConnectWrap.afterConnect [as oncomplete] (node:net:1555:16)
[3] Emitted 'error' event on WebSocket instance at:
[3]     at emitErrorAndClose (C:\Users\yogie\Documents\GitHub\IoTTestingMicro\Payload\node_modules\ws\lib\websocket.js:1035:13)
[3]     at ClientRequest.<anonymous> (C:\Users\yogie\Documents\GitHub\IoTTestingMicro\Payload\node_modules\ws\lib\websocket.js:880:5)
[3]     at ClientRequest.emit (node:events:517:28)
[3]     at Socket.socketErrorListener (node:_http_client:501:9)
[3]     at Socket.emit (node:events:517:28)
[3]     at emitErrorNT (node:internal/streams/destroy:151:8)
[3]     at emitErrorCloseNT (node:internal/streams/destroy:116:3)
[3]     at process.processTicksAndRejections (node:internal/process/task_queues:82:21) {
[3]   errno: -4039,
[3]   code: 'ETIMEDOUT',
[3]   syscall: 'connect',
[3]   address: '10.42.0.75',
[3]   port: 8484
[3] }
[3]
[3] Node.js v18.18.0
```

**Figure 12: Erreur de connexion avec l'application WIMP**

Finalement, pour le test de système, tout fonctionne correctement, et l'intégration entre le frontend et le backend se déroule sans accroc. Le frontend ordonne désormais correctement les résultats de tests reçus du service "ExecuteTest", garantissant une présentation claire et organisée des données. Le programme fonctionne efficacement en tant qu'entité complète, et l'ensemble du système opère sans problèmes majeurs.

Lors de l'envoi d'un payload via l'interface, le programme a été ajusté pour vérifier que la connexion est bien établie avant d'envoyer les informations de test. Cela garantit que les requêtes sont envoyées de manière synchronisée et que les résultats sont traités correctement. Grâce à ces ajustements, le programme fonctionne de manière fluide, sans erreurs ni interruptions.

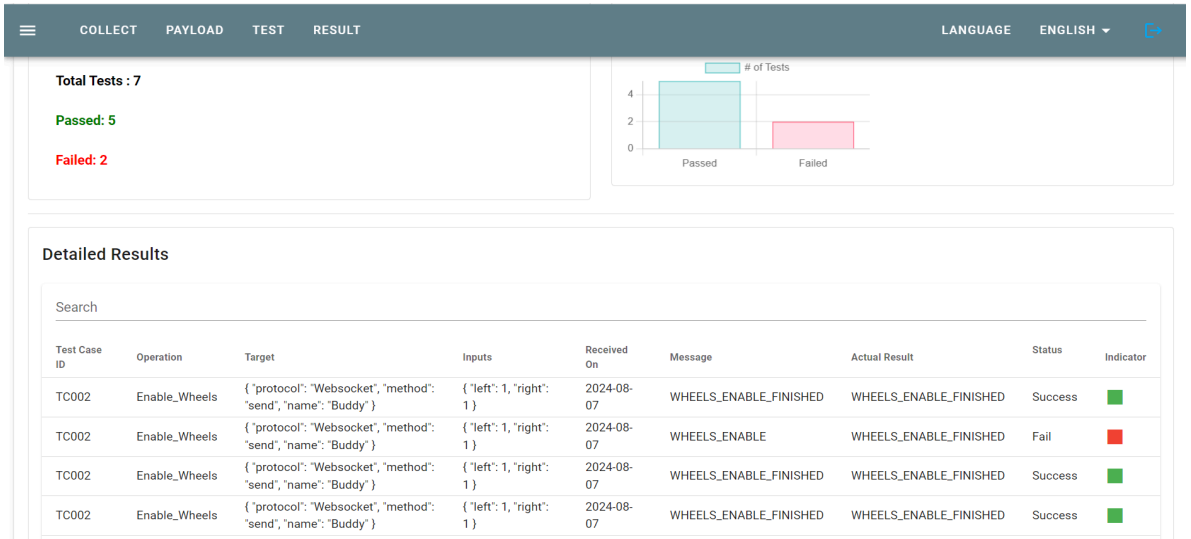


**Figure 13: Résultats de tests affichés dans l'interface**

# Conclusion

Dans le cadre de notre projet de fin d'études intitulé "IoT Systems Testing", nous avons été chargés d'assister l'équipe Ptidej de Concordia dans le test de leur application "Where is my Professor?" (WIMP). Constatant certaines limitations des outils de test existants, nous avons décidé de développer notre propre solution, reposant sur une architecture microservice, pour envoyer et sauvegarder des commandes de test à l'application WIMP. En plus de cette fonction principale, notre application devait inclure un système de gestion des connexions pour les utilisateurs. Par la suite, nous avons adapté notre solution pour supporter différents types de connexions, tels que CoAP et MQTT.

Afin de définir les objectifs et la méthode de conception de notre application, nous avons d'abord discuté avec un membre de l'équipe Ptidej pour confirmer les détails de la construction. Nous nous sommes ensuite familiarisés avec les outils utilisés par l'application, en particulier le robot "Buddy", à travers des interactions et la consultation de manuels d'instructions. Une fois les bases posées, notre équipe s'est répartie les tâches : l'un travaillait sur le frontend tandis que l'autre se concentrait sur le backend. Une fois ces deux parties finalisées, nous avons procédé à leur intégration pour permettre leur interaction.

En ce qui concerne nos objectifs, la majorité d'entre eux ont été atteints avec succès. Notre application est capable d'analyser efficacement les payloads et de les transmettre à l'application WIMP, qui exécute ensuite les commandes et les enregistre en tant que résultats de test. Bien que l'objectif concernant le reçu des résultats n'ait pas été entièrement accompli, tel que décrit dans la section 1.3, notre application peut tout de même recevoir les résultats des tests de WIMP, bien que cela nécessite l'envoi d'une requête pour les récupérer, avant de les stocker dans notre propre base de données.

Notre solution suit l'architecture microservice, telle que définie par les sources consultées en ligne. Chaque composant du logiciel peut être exécuté indépendamment ou en conjonction avec d'autres, les sous-services actifs continuant de fonctionner de manière fluide. Cela a également facilité la correction des erreurs, en permettant d'identifier rapidement la source du problème. Nous avons réussi à créer une version fonctionnelle de notre programme en environ trois mois, bien que certaines améliorations n'aient pas pu être mises en œuvre par manque de temps. Le seul objectif non atteint est celui lié au critère de réussite final. Cet échec est principalement dû à des facteurs externes, notamment les difficultés rencontrées par l'équipe Ptidej à fournir une connexion sécurisée permettant

de communiquer simultanément avec WIMP et nos bases de données. Cela a donc limité notre capacité à tester nos cas de test à un taux de réussite optimal.

En ce qui concerne les résultats des tests, la plupart des fonctions de notre application ont été validées avec succès, en particulier celles liées à l'interaction entre les différents composants. Toutefois, les tests d'envoi de payloads vers WIMP ou d'autres applications IoT ont donné des résultats moins concrets, principalement en raison de contraintes externes.

Pour l'avenir, il serait intéressant de tester notre application avec d'autres logiciels nécessitant différents types de connexions. Cela permettrait d'évaluer la polyvalence de notre solution et son adaptabilité à divers systèmes IoT.

# Bibliographies

**Tremblay, M., & Gagnon, F. (2020)**. *Cadres de test pour les systèmes IoT au Québec : Enjeux et solutions*. Revue québécoise des sciences et technologies de l'information, 45(2), 112-128.

**IBM.** 2018. « Microservices : Une approche pour concevoir des systèmes logiciels modulaires ». In IBM. En ligne. https://www.ibm.com/cloud/learn/microservices . Consulté le 2 juin 2024.

**Automatisation Québec.** 2021. « Les meilleures pratiques pour tester les systèmes IoT ». In Automatisation Québec. En ligne. https://www.automatisationquebec.com/tests-iot . Consulté le 17 août 2024.

**Vue.js.** 2021. « Guide de Vue.js pour les développeurs IoT ». In Vue.js. En ligne. https://vuejs.org/guide . Consulté le 10 mai 2024.

**Vuetify.** 2021. « Vuetify: Material Design Component Framework ». In Vuetify. En ligne. https://vuetifyjs.com/en/. Consulté le 15 mai 2024.

**N-iX.** Année de publication inconnue. « Expert Steps to Successful IoT Test Automation. In N-iX. En ligne. https://www.n-ix.com/expert-steps-to-successful-iot-test-automation/. Consulté le 25 juin 2024.

**MongoDB.** 2021. « Utilisation de MongoDB pour les applications IoT ». In MongoDB. En ligne. https://www.mongodb.com/industries/internet-of-things. Consulté le 12 mai 2024.

**MongoDB.** 2021. « Utilisation de MongoDB pour les applications IoT ». In MongoDB. En ligne. https://www.mongodb.com/fr-fr/atlas . Consulté le 12 mai 2024.

Onukwube, Eze. 2024. « 18 Best IoT Testing Tools For QA Teams In 2024 ». In QA Lead. En ligne. <https://theqalead.com/tools/best-iot-testing-tools/>. Consulté le 2 juillet 2024.

WebdriverIO. 2023. « WebdriverIO : Navigateur de nouvelle génération et framework de test d'automatisation mobile pour Node.js». In WebdriverIO. En ligne. <https://webdriver.io/fr/>. Consulté le 17 juillet 2024.

Code With Mmak. Année de publication inconnue. « WebdriverIO – A Step-by-Step Comprehensive Guide with Examples ». In codewithmmak. En ligne. <https://www.codewithmmak.com/webdriverio-a-step-by-step-comprehensive-guide-with-examples/ >. Consulté le 17 juillet 2024.

OpenJS Foundation. 2024. « Mocha : simple, flexible, fun ». In mochajs. En ligne. <https://mochajs.org/>. Consulté le 17 juillet 2024.