Dédicaces

Remerciements

Résumé

Abstract

Liste des abréviations

Abréviation	**Désignation**
API	Application Programming Interface
CPU	Central Processing Unit
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JPA	Java Persistence API
LLM	Large Language Model
PDF	Portable Document Format
RAG	Retrieval-Augmented Generation
SARL	Société À Responsabilité Limitée
SQL	Structured Query Language
SRP	Single Responsibility Principle
UML	Unified Modeling Language
URL	Uniform Resource Locator

Table des figures

Liste des tableaux

Table des matières

Introduction

Chapitre 1

Contexte du Projet

1.1 Projet de Fin d'Études

Dans le cadre de notre formation en Génie Informatique à l'École des Hautes Études d'Ingénierie d'Oujda (EHEIO), nous devons réaliser un Projet de Fin d'Études (PFE), visant à consolider et approfondir les compétences acquises durant notre parcours. Ce stage représente une étape cruciale, permettant de transposer les connaissances théoriques dans un environnement professionnel concret.

L'objectif principal est de se familiariser avec les réalités du marché du travail, particulièrement dans le domaine de l'informatique, en perpétuelle évolution. Ce projet offre ainsi l'opportunité d'appliquer nos acquis académiques à des problématiques réelles, tout en développant une approche pratique et méthodique de la gestion de projets technologiques.

Ce stage s'est déroulé dans une entreprise adoptant une méthodologie Scrum, l'une des approches Agile les plus répandues dans le secteur informatique. Cette immersion professionnelle a été l'occasion découvrir le fonctionnement d'une équipe projet en conditions réelles, d'appliquer les principes Agile (itérations, sprints, réunions quotidiennes) dans un cadre professionnel, et de collaborer avec des experts et assimiler les bonnes pratiques en gestion de projets logiciels.

L'utilisation de Scrum a renforcé ma compréhension des processus modernes de développement, tout en améliorant mes capacités d'adaptation et de travail en équipe. Cette expérience a été déterminante pour affiner ma vision du métier d'ingénieur informatique et préparer mon intégration dans le monde professionnel.

Dans ce chapitre, nous commencerons par présenter l'entreprise d'accueil, avant de procéder à une description détaillée des besoins du projet. Cette description comprendra l'identification du problème posé, l'analyse des besoins fonctionnels et non fonctionnels, ainsi que les solutions envisagées pour y répondre.

1.2 Entreprise d'accueil

Ce stage a été réalisé au sein de l'entreprise Algolus, située à Oujda, spécialisée dans les solutions innovantes en intelligence artificielle. Démarré le 24 février 2024, il m'a permis de m'immerger dans un environnement professionnel exigeant, où j'ai pu collaborer avec des experts en IA et en ingénierie logicielle.

FIGURE 1.1 – *Logo de Algolus*

1.2.1 Description de l'entreprise

Algolus est une agence web marocaine, créée en 2020, spécialisée dans la conception et le développement de solutions informatiques adaptées aux besoins des clients. Elle s'engage à offrir à ses clients une communication en ligne efficace et sur mesure.

Ses prestations incluent :

- Création et gestion de sites web (dynamiques, statiques, e-commerce, CMS)
- Développement d'applications web (mode hybride)
- Stratégie digitale complète : infographie, publicité en ligne, marketing digital, community management, E-réputation.

1.2.2 Fiche technique de l'entreprise

Le tableau 1.1 récapitule la fiche technique de l'entreprise Algolus :

TABLE 1.1 – *Fiche technique de l'entreprise Algolus*

Dénomination sociale	Algolus
Date de création	07/10/2020
Forme juridique	SARL
Capital	100.000 Dh
Chiffre d'affaires	Indisponible
Activités	Développement informatique et marketing digital
Effectif	10
Dirigeant	Radwane BERAHIOUI
Coordonnées	+212 6644 35967 Redwan.Berahioui@algolus.ma www.algolus.ma IMMEUBLE OUASSIM, Bd Mohammed VI, Oujda 60000

1.2.3 Organigramme de l'entreprise

La figure 1.2 présente l'organigramme de l'entreprise Algolus :

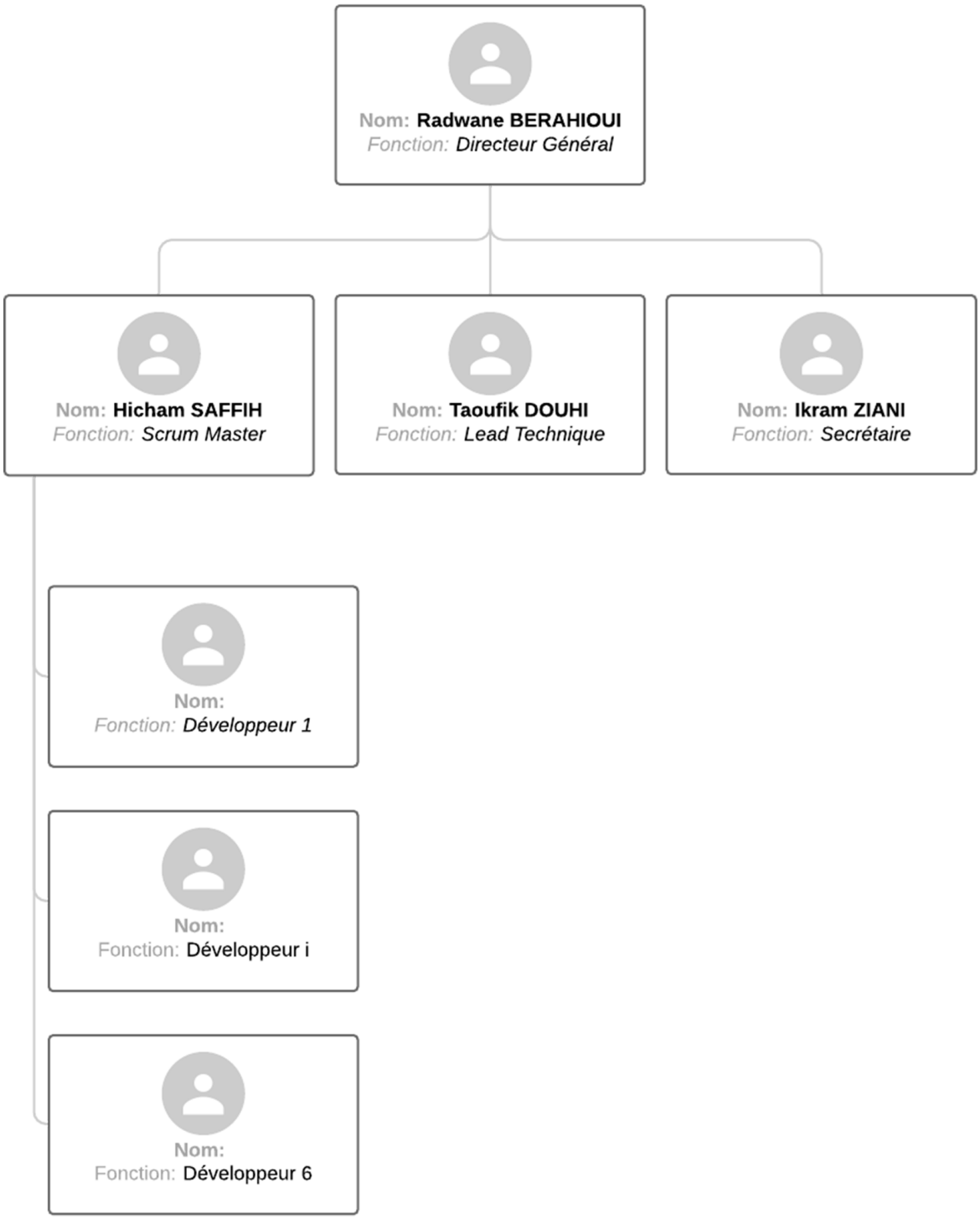


FIGURE 1.2 – *Organigramme de l'entreprise Algolus*

1.3 Description des besoins

1.3.1 Problème

Dans le cadre du développement logiciel la détection et la correction des erreurs représentent un défi majeur, notamment en raison de la diversité des sources d'anomalies (logs, stack traces, captures d'écran, retours utilisateurs, etc.) et de la complexité croissante des applications. Les méthodes traditionnelles de débogage reposent souvent sur une analyse manuelle, ce qui est chronophage et sujet à des erreurs humaines. De plus, les solutions existantes peinent à offrir une approche générique et intelligente pour interpréter ces anomalies et proposer des correctifs pertinents.

Ce défi prend une dimension particulière dans le cadre des activités de Algolus. La complexité des systèmes gérés par l'entreprise amplifie les difficultés de diagnostic des anomalies. Les équipes techniques consacrent actuellement un volume considérable de leurs ressources temporelles à l'analyse manuelle des incidents, retardant d'autant les mises en production. Par ailleurs, la variété des clients et des cas d'usage entraîne une hétérogénéité des remontées d'erreurs (rapports techniques détaillés pour les clients corporate vs. simples captures d'écran pour les utilisateurs finaux), ce qui rend inefficaces les outils de monitoring conventionnels utilisés jusqu'à présent. Ce constat a motivé l'entreprise à explorer des solutions d'IA générative capables d'unifier l'interprétation des anomalies.

1.3.2 Les besoins fonctionnels

Les besoins fonctionnels définissent les actions spécifiques que le système doit accomplir pour répondre aux exigences métier. Ils décrivent le "quoi", qu'est ce que le système doit faire, sous la forme de fonctionnalités concrètes, de processus et d'interactions avec l'utilisateur. Ces exigences sont formulées par les parties prenantes, à savoir les clients, les utilisateurs, et l'équipe produit, et servent de base à la conception des cas d'usage et des scénarios de test.

Pour garantir que le système de diagnostic d'erreurs réponde efficacement aux attentes des utilisateurs et des équipes techniques, nous avons identifié les besoins fonctionnels suivants :

1. **Collecte et Pré-traitement des Données** : Extraction automatique des erreurs et des anomalies des systèmes à partir de : stacktraces, parsing des logs, captures d'écran, retours utilisateurs.
2. **Analyse et Compréhension** : Analyse sémantique de retours d'erreurs, enrichissement contextuel : requêtage d'une base de connaissances (documentation technique, correctifs historiques) via RAG (Retrieval-Augmented Generation).

3. **Génération de Solutions** : Explication en langage naturel des causes racines, génération de correctifs (ex : snippets de code, étapes de résolution).
4. **Interfaces utilisateurs** : Soumission des erreurs via des formulaires web pour uploader des stacktraces et des captures d'écran, visualisation des résultats : Dashboard interactif (erreurs en cours, historiques, statistiques).

1.3.3 Les besoins non fonctionnels

Les besoins non fonctionnels caractérisent le "comment", c'est à dire comment le système doit fonctionner, en précisant ses contraintes de qualité, de performance et d'infrastructure. Contrairement aux besoins fonctionnels, ils ne décrivent pas des fonctionnalités mais des critères tels que la rapidité, la sécurité, la scalabilité ou la facilité de maintenance. Leur respect est essentiel pour assurer la robustesse et l'efficacité du système en conditions réelles.

Pour garantir une intégration harmonieuse dans l'écosystème existant et une expérience utilisateur optimale, les besoins non fonctionnels suivants ont été définies :

1. **Performances** : Temps de réponse optimisé, et scalabilité : Support de plusieurs requêtes simultanées.
2. **Intégration et Interopérabilité** : API REST : Endpoints standardisés et format de réponse avec schéma cohérent, support offline : fonctionnement local avec Ollama.
3. **Sécurité et Confidentialité** : Protection des données par chiffrement des échanges et anonymisation des logs utilisateurs (RGPD), et authentification : JWT pour l'accès aux APIs sensibles.
4. **Expérience Utilisateur** : Ergonomie : interface intuitive, Dark/Light mode et thèmes accessibles.

1.3.4 Solutions envisagées

Ce projet vise à développer une application intelligente et modulaire permettant de détecter et de corriger automatiquement les anomalies logicielles. Les objectifs spécifiques incluent :

- Détection avec un taux de réussite d'au moins 80% les anomalies logicielles sur des sources multimodales.
- Proposition automatique des correctifs pertinents dans plus de 70% des cas.
- Optimisation du temps moyen de résolution d'erreurs de 30% par rapport aux méthodes manuelles.

Pour atteindre ces objectifs, le projet s'appuie sur une architecture innovante :

1. **Analyse Multimodale des Erreurs** : Implémenter un système capable d'interpréter des données hétérogènes (stack traces, logs texte, captures d'écran, etc.), et utiliser

des techniques de RAG pour enrichir les requêtes avec une base de connaissances (documentation technique, résolutions d'erreurs courantes).

2. **Génération Automatique de Correctifs** : Exploiter des LLMs (via Ollama) pour suggérer des corrections précises et contextualisées.
3. **Intégration et Scalabilité** : Développer un backend Spring Boot flexible, couplé à LangChain4J pour orchestrer les appels IA, et un système de gestion de base de données qui prend en charge les bases de données vectorielles, comme PostgeSQL, et permettre une extension future via des connecteurs pour différents outils de monitoring.
4. **Optimisation et Évaluation** : mesurer l'efficacité du système via des métriques de précision (taux de détection, pertinence des correctifs), et effectuer un Benchmark : comparaison sur des jeux de données communs.

Chapitre 2

Analyse fonctionnelle et modélisation

2.1 Importance de l'analyse fonctionnelle

L'analyse fonctionnelle constitue une étape clé dans tous projets de développement informatique, car elle permet de bien comprendre les besoins du client et les contraintes du système à réaliser. Elle sert à identifier les fonctionnalités attendues, à détecter les éventuelles incohérences et à poser les bases d'une conception solide. Une analyse fonctionnelle bien menée réduit considérablement les risques d'erreurs en phase de développement, facilite la planification du travail et améliore la qualité globale du produit final, ce qui lui rend essentielle pour assurer la réussite du projet.

Dans ce chapitre nous présentons une introduction à la modélisation UML, en rappelant ses principes fondamentaux et son utilité dans le développement logiciel. Par la suite nous exposerons les différents types de diagrammes utilisés dans notre étude, à savoir le diagramme de cas d'utilisation, les diagrammes de séquences, ainsi que le diagramme de classes.

2.2 Unified Modeling Language

Dans le cadre d'un projet de développement informatique la modélisation UML (Unified Modeling Language) joue un rôle essentiel en facilitant la compréhension, la conception et la communication autour du système à développer. UML propose un ensemble de diagrammes normalisés qui permettent de représenter visuellement les différentes dimensions d'un logiciel, telles que la structure, le comportement et les interactions entre les composants.

L'utilisation des diagrammes UML comme les diagrammes de cas d'utilisation, de classes et de séquences permet de clarifier les besoins fonctionnels et non fonctionnels dès les premières phases du projet, de favoriser une meilleure communication entre les développeurs, les analystes et les clients, de détecter de façon précoce les incohérences ou erreurs potentielles dans la conception, et aussi de servir de documentation technique

structurée pour le développement, les tests et la maintenance future du logiciel.

Ainsi, UML constitue un outil précieux pour assurer la qualité, la cohérence et la pérennité d'un projet informatique, en apportant une vision globale et partagée du système.

2.3 Diagramme de cas d'utilisation

2.3.1 Définition

Un diagramme de cas d'utilisation est une représentation visuelle des interactions entre les acteurs (utilisateurs, systèmes) et les fonctionnalités d'une application. Il identifie les besoins métiers sous forme d'actions (cas d'utilisation) et montre qui fait quoi, sans entrer dans les détails techniques.

2.3.2 Acteurs

Dans un diagramme de cas d'utilisation, les acteurs sont les entités qui interagissent avec le système pour accomplir un objectif précis. Un acteur peut être primaire (s'il est déclencheur d'un cas d'utilisation) ou secondaire (intervient dans un cas d'utilisation mais ne le déclenche pas). D'une autre part, un acteur peut être humain ou bien un acteur système.

Trois types d'acteurs sont impliqués dans notre cas :

- **Utilisateur** : peut être un développeur ou un testeur qui rapporte une erreur, et peut interagir via une API REST ou bien une interface web.
- **Administrateur du système** : responsable de la mise à jour des connaissances du système et de la configuration des modèles.
- **Système** : le moteur de traitement intelligent, responsable d'analyser les anomalies, et de proposer des correctifs appropriés.

2.3.3 Diagramme de cas d'utilisation

La figure 2.1 présente le diagramme de cas d'utilisation de notre application :

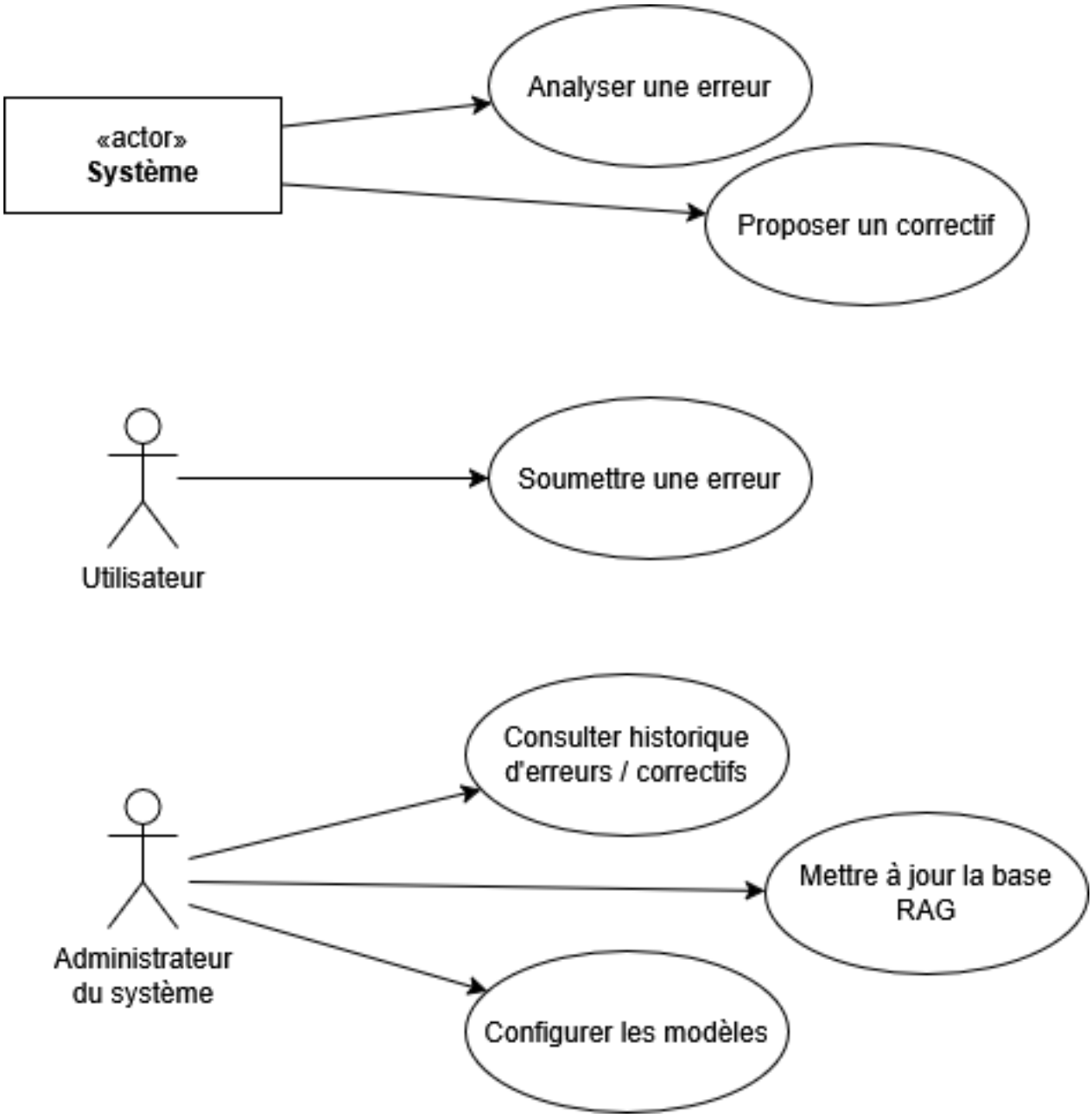


FIGURE 2.1 – *Diagramme de cas d'utilisation*

2.4 Diagrammes de séquences

2.4.1 Définition

Un diagramme de séquences est un type de diagramme UML, utilisé pour modéliser les interactions entre les différents objets ou composants d'un système dans un scénario précis. Il met en évidence l'ordre chronologique des messages échangés entre les acteurs et les objets, les interactions dynamiques entre les éléments du système, et la durée de vie

des objets participant au scénario.

2.4.2 Diagramme de séquences principal

Cette application d'analyse d'erreurs techniques repose sur une architecture modulaire, où chaque composant joue un rôle précis dans le traitement des requêtes. Voici une présentation des éléments clés qui permettent au système de comprendre, contextualiser et répondre aux problèmes soumis par les utilisateurs :

- **Utilisateur** : L'Utilisateur constitue le point de départ du système. Ce composant représente l'acteur humain qui interagit avec l'application via une interface web ou des appels API. Il soumet des requêtes contenant des stacktraces d'erreur et éventuellement des captures d'écran.
- **LLM (Large Language Model)** : Est un modèle d'intelligence artificielle entraîné sur des volumes massifs de données textuelles, capable de comprendre, générer et manipuler du langage naturel de manière contextuelle. Basé sur des architectures de deep learning (comme les transformers), il excelle dans des tâches variées (réponse aux questions, traduction, synthèse de texte, etc.) en prédisant des séquences de mots probabilistes. Contrairement aux systèmes traditionnels, un LLM ne suit pas de règles prédéfinies, mais apprend des motifs linguistiques à partir de ses données d'entraînement.
- **RAG (Retrieval-Augmented Generation)** : Est une architecture hybride combinant un système de recherche d'information (retrieval) et un modèle de génération de langage (LLM). Contrairement à un LLM standard qui repose uniquement sur ses connaissances internes, le RAG enrichit ses réponses en recherchant des documents pertinents dans une base de connaissances externe, et en générant des réponses contextualisées à partir de ces sources. son plus grand avantage est la mise à jour sans réentraînement du LLM, via la base de connaissances.
- **Agent AI** : Sert de chef d'orchestre principal dans l'architecture. C'est un service intelligent qui coordonne l'ensemble du processus d'analyse. Il reçoit les requêtes brutes de l'utilisateur, les enrichit en combinant plusieurs techniques avancées comme le RAG et la mémoire conversationnelle, puis les présente au modèle de langage sous une forme optimale.

  Parmi les atouts majeurs d'un Agent AI sa capacité à configurer dynamiquement les trois parties constituant le prompt envoyé au LLM :
  - **System Message** : définit le comportement global attendu du modèle (rôle, ton, contraintes, format de la réponse), en fixant un cadre pour l'interprétation des requêtes.
  - **User Input** : correspond à la requête explicite formulée par l'utilisateur, exprimant son besoin ou sa question.

— **Few Shot Examples** : Quelques exemples de paires question/réponse (ou tâche/résultat), avant la question réelle de l'utilisateur, servant à orienter le comportement du modèle sans avoir à l'entraîner à nouveau.

- **ChatModel** : Incarne le moteur de génération de langage naturel. Ce composant spécifique, configuré pour utiliser des modèles locaux ou externes, transforme les prompts structurés en analyses techniques détaillées.

  Le ChatModel permet de configurer plusieurs propriétés du LLM, citons les plus importantes :

  — L'URL de base étant le point d'accès à l'API du modèle
  — Le nom du modèle de langage à utiliser
  — La Température, une propriété qui détermine le degré de créativité du LLM à formuler ses réponses, 0 étant le plus précis, et 1 le plus créatif.
  — Le Timeout, qui est le délai maximal de réponse avant échéance.
- **ChatMemory** : est un composant logiciel conçu pour conserver l'historique des échanges dans un système conversationnel (comme un chatbot), permettant ainsi de maintenir le contexte entre les messages et d'offrir des réponses plus cohérentes et personnalisées.

Le diagramme de séquences dans la figure 2.2 décrit le flux de messages entre ces composants.

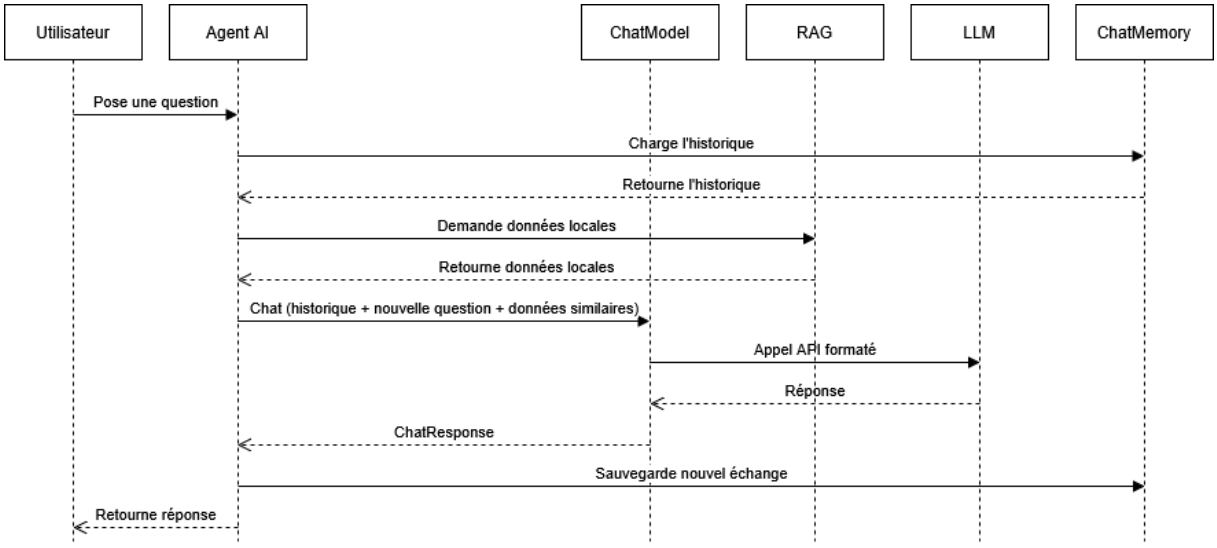


FIGURE 2.2 – *Diagramme de séquences décrivant le fonctionnement d'un agent AI*

2.4.3 Diagramme de séquences spécifique : RAG

Nous allons creuser un peu dans les composants du système RAG, voici une énumération de ses composants avec chacun son rôle :

- **Resource** : Constitue la matière première du système RAG. Il s'agit de la source originelle des données qui alimenteront la base de connaissances. Ces ressources peuvent prendre diverses formes : fichiers PDF contenant la documentation technique, pages web de référence, extraits de bases de données, ou tous autres supports contenant des informations pertinentes.
- **Tokenizer** : Joue un rôle fondamental dans le prétraitement du texte. Ce composant décompose le contenu textuel en unités significatives appelées tokens, "un token peut correspondre à un mot entier, un sous-mot ou même un caractère individuel selon la méthode employée". Des algorithmes avancés comme ceux proposés par HuggingFace permettent une optimale qui préserve le sens tout en gérant les particularités linguistiques. La *tokenisation* joue un rôle crucialecar elle influe directement la qualité des embeddings générés ultérieurement.
- **DocumentParser** : Assure la transformation des ressources brutes en documents structurés. Ce composant doit comprendre divers formats de fichiers (PDF, HTML, Markdown, etc.) et en extraire le contenu textuel significatif tout en conservant les métadonnées importantes. Des bibliothèques spécialisées comme sont souvent employées pour cette tâche complexe. Le DocumentParser nettoie également le texte en supprimant les éléments non pertinents (en-têtes, pieds de page, balises HTML) pour ne conserver que l'information essentielle.
- **Document** : Représente la forme normalisée et standardisée des informations après traitement. Chaque document contient non seulement le texte brut nettoyé, mais aussi des métadonnées descriptives (titre, auteur, date de création, source) qui faciliteront son identification et son utilisation ultérieure. Un identifiant unique est attribué à chaque document pour permettre son suivi tout au long du pipeline. Cette structuration rigoureuse est essentielle pour maintenir la cohérence des données dans les étapes suivantes du processus RAG.
- **EmbeddingModel** : Est au cœur de la transformation sémantique du système. Ce modèle sophistiqué convertit le texte en représentations vectorielles denses (embeddings) qui capturent le sens profond des contenus. Des modèles sont spécialisés dans cette tâche produisent des vecteurs où la similarité spatiale correspond à la similarité sémantique. La qualité de l'EmbeddingModel détermine directement la capacité du système à retrouver des documents pertinents pour une requête donnée.
- **EmbeddingStoreIngestor** : Orchestre le processus complet d'indexation des documents. Ce composant supervise plusieurs opérations critiques : il applique la tokenisation et la segmentation des textes, déclenche la génération des embeddings via l'EmbeddingModel, et gère le stockage final dans l'EmbeddingStore.
  L'EmbeddingStoreIngestor implémente souvent des stratégies de traitement par lots pour optimiser les performances et peut gérer des pipelines complexes de prétraitement avant la vectorisation.

- **EmbeddingStore** : Sert de mémoire à long terme au système RAG. Cette base de données vectorielle spécialisée stocke les embeddings générés et permet des recherches rapides de similarité. L'EmbeddingStore doit supporter des opérations massives d'insertion tout en maintenant des temps de réponse faibles pour les requêtes.
- **Retriever** : Est le composant qui établit le pont entre les questions des utilisateurs et la base de connaissances. Lors d'une requête, le Retriever transforme d'abord la question en embedding, puis recherche dans l'EmbeddingStore les documents dont les vecteurs sont les plus proches. Ce composant implémente des algorithmes de similarité vectorielle (cosine similarity par exemple) et peut être finement paramétré (nombre de résultats retournés, seuil de similarité minimal). Le Retriever joue ainsi un rôle déterminant dans la pertinence des résultats fournis au LLM.

Le diagramme de séquences dans la figure 2.3 résume ce processus.

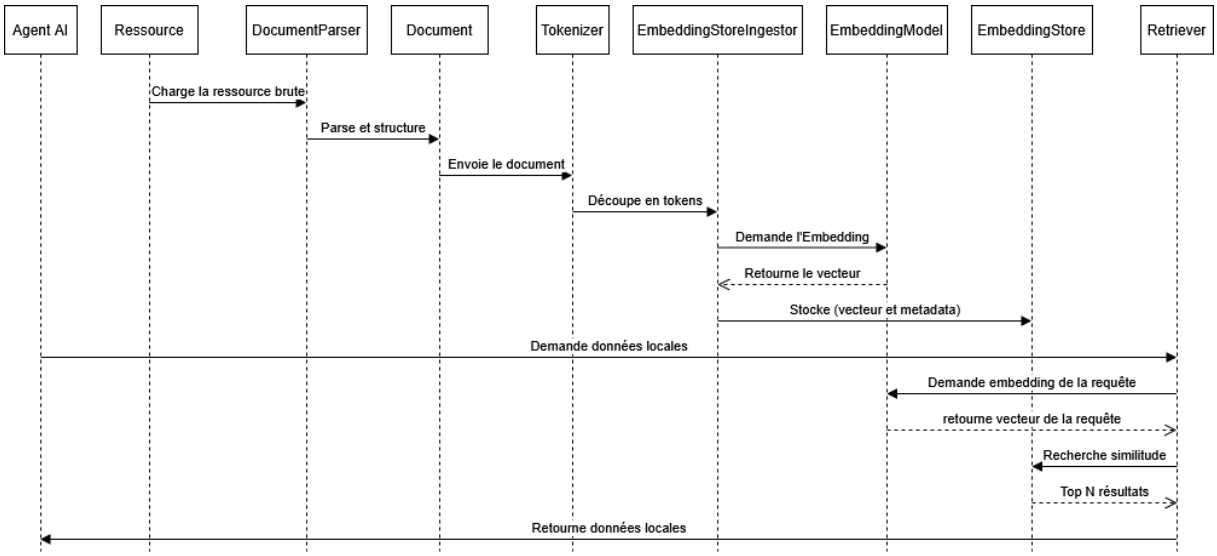


FIGURE 2.3 – *Diagramme de séquences décrivant le fonctionnement d'un RAG*

Chapitre 3

Réalisation

Ce chapitre présente la phase de concrétisation du projet, où les choix techniques et architecturaux se traduisent en une implémentation fonctionnelle. Il décrit l'environnement de développement (outils, frameworks, librairies, etc.), l'architecture logicielle retenue et son adéquation avec les besoins, les défis techniques rencontrés et les solutions apportées, et les composants clés implémentés avec des extraits de code significatifs, et des captures d'écran illustrant les résultats obtenus.

3.1 Stack technique

3.1.1 Langages

Java

Java est un langage de programmation orienté objet, robuste et multiplateforme, largement utilisé dans le développement d'applications d'entreprise. Sa forte typographie, sa gestion automatique de la mémoire (via le garbage collector) et son écosystème riche (bibliothèques, frameworks) en font un choix idéal pour les systèmes backend complexes.

3.1.2 Frameworks

Spring

Spring est un framework modulaire pour Java, simplifiant le développement d'applications grâce à l'inversion de contrôle (IoC) et la programmation orientée aspect (AOP).

Spring Boot

Spring Boot étend Spring en fournissant des configurations automatiques, un serveur embarqué (Tomcat, Netty) et des outils clés en main (Spring Data, Spring Security), permettant de créer des applications standalone rapidement.

LangChain4j

LangChain4J est une bibliothèque Java inspirée de LangChain (Python), conçue pour intégrer facilement des LLMs (Modèles de Langage) dans des applications. Elle offre des abstractions pour la gestion des prompts, le RAG, les appels aux modèles (OpenAI, Ollama, etc.), et la connexion à des bases de données vectorielles.

3.1.3 Bibliothèques

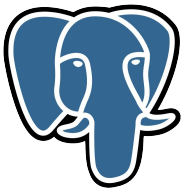
Apache Commons

Apache Commons est une bibliothèque Java open-source fournissant des composants réutilisables pour simplifier le développement. Dans ce projet, elle sert à combler des besoins techniques récurrents avec des solutions optimisées et robustes.

3.1.4 Systèmes de gestion de bases de données

PostgreSQL

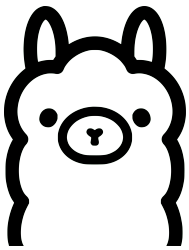
PostgreSQL est un système de gestion de base de données relationnelle (SGBDR) open-source, robuste et extensible. Dans le cadre de ce projet, il joue un rôle central pour stocker et gérer les données structurées nécessaires au bon fonctionnement de l'application, et fournit des plugins pour l'IA, notamment PgVector, qui gère les bases de données vectorielles.



3.1.5 Outils et environnement

Ollama

Ollama est un outil open-source permettant d'exécuter localement des LLMs (comme Llama 3, Mistral, Gemma) sans dépendre d'une API externe. Il est idéal pour prototyper des solutions IA offline, contrôler les coûts et la confidentialité des données, et personnaliser finement les modèles via des modelfiles.



Maven

Outil de build automatisé pour projets Java, qui gère Les dépendances (téléchargement auto), le packaging (JAR/WAR), et les cycles de compilation/test.

IntelliJ IDEA

IntelliJ IDEA est un IDE puissant pour Java/Kotlin, développé par JetBrains. Ses avantages incluent une analyse intelligente du code (suggestions, détection d'erreurs), une intégration native avec Spring Boot et Maven/Gradle, des outils pour le débogage, le profiling et les tests, et des extensions pour l'IA (ex : GitHub Copilot).

Git

Git est un système de contrôle de version distribué, essentiel pour le développement collaboratif. Il permet de suivre les modifications du code source, de gérer les branches, et de fusionner les travaux de plusieurs contributeurs. Grâce à des plateformes comme GitHub, il facilite le partage et la revue de code. Son utilisation améliore la traçabilité, la qualité et la productivité dans les projets logiciels.

3.2 Architecture technique du projet

Le projet repose sur une architecture modulaire et évolutive, construite autour des meilleures pratiques du développement Java moderne, de l'IA générative et de l'ingénierie logicielle. Il s'appuie sur les technologies suivantes :

- **Spring Boot** : Est le framework retenu pour le développement de la couche back-end. Il s'agit d'un choix stratégique largement justifié par les besoins du projet en termes de performance, de maintenabilité et d'intégration avec des composants d'intelligence artificielle.

  Spring Boot permet de structurer l'application de manière modulaire, en séparant clairement les responsabilités (contrôleurs, services, configuration, etc.). Cette organisation favorise une bonne lisibilité du code et facilite son évolution.

  L'application est également portable, dans la mesure où elle peut être conditionnée sous forme de JAR exécutable et déployée facilement sur tout environnement compatible Java, sans dépendance à un serveur externe.

  Un autre atout majeur est le caractère évolutif de Spring Boot : il s'intègre naturellement avec des bibliothèques telles que LangChain4j ou des bases de données comme PostgreSQL, ce qui permet d'ajouter de nouvelles fonctionnalités (IA, recherche vectorielle, mémoire contextuelle) sans remise en cause de l'existant.

  Le framework est aussi testable : il propose des outils natifs pour la réalisation de tests unitaires et d'intégration, garantissant la qualité et la stabilité du code produit.

  Enfin, Spring Boot est hautement extensible. Si le projet venait à croître en complexité, il serait tout à fait envisageable de faire évoluer l'architecture vers un modèle microservices avec des outils comme Spring Cloud.
- **LangChain4j** : Pour permettre l'analyse intelligente des erreurs à l'aide d'un modèle de langage (LLM), le projet s'appuie sur la bibliothèque LangChain4j, une adaptation Java du framework LangChain initialement développé pour Python. Ce composant joue un rôle central dans l'intégration des fonctionnalités d'intelligence artificielle générative.

LangChain4j facilite la mise en œuvre d'un mécanisme de RAG (Retrieval-Augmented Generation), en combinant génération de texte via un LLM et récupération de documents pertinents à partir d'une base vectorielle. Il s'intègre naturellement avec un modèles de langage, et avec des composants tels que la mémoire de conversation, le modèle d'embedding et le store d'embeddings.

L'utilisation de LangChain4j rend l'application extensible et modulaire, car ses composants (LLM, mémoire, embeddings, etc.) sont interchangeables via des interfaces. Elle offre également un haut niveau de configurabilité, permettant d'adapter dynamiquement les modèles utilisés, la taille de la mémoire contextuelle ou encore les critères de pertinence documentaire.

Enfin, son intégration avec Spring Boot via des beans injectables simplifie grandement sa mise en œuvre dans l'architecture globale du projet. Cela permet d'enrichir les traitements métier avec une couche d'IA tout en conservant la lisibilité et la testabilité du code.

- **Ollama** : Pour exécuter les modèles de langage en local sans dépendre de services cloud externes, le projet intègre Ollama, une plateforme légère permettant de servir des LLM open-source tels que Mistral, Llama3 ou Qwen qui offre des versions multimodales. Ollama agit comme un point d'accès HTTP local à un modèle de génération de texte, que LangChain4j peut interroger de manière transparente.

  Ce choix présente plusieurs avantages : tout d'abord, il rend l'application autonome et portable, car aucun appel à une API cloud (comme OpenAI ou Hugging Face) n'est requis. Cela permet un déploiement sur des machines locales ou en environnement isolé (on-premise), tout en respectant les contraintes de confidentialité des données.

  Grâce à une configuration centralisée (adresse du serveur, modèle utilisé, température, etc.), Ollama est également hautement configurable. Son intégration dans le projet se fait via des beans Spring instanciés dynamiquement dans la classe de configuration (AiConfig), ce qui permet d'adapter ou changer le modèle utilisé sans modifier la logique métier.

  Enfin, en travaillant de concert avec LangChain4j, Ollama permet la génération de réponses contextualisées et pertinentes, en tenant compte des documents récupérés et des interactions passées. Cela renforce la capacité de l'application à fournir des analyses d'erreurs enrichies, précises et directement exploitables.
- **PostgreSQL** : Utilisé comme système de gestion de base de données relationnelle, avec une orientation spécifique vers le stockage vectoriel, dans le cadre de l'indexation et de la recherche de documents sémantiques. Grâce à l'extension pgvector, PostgreSQL devient capable de stocker des vecteurs d'embedding et d'effectuer des recherches de similarité, essentielles dans une approche RAG (Retrieval-Augmented Generation).

  Le choix de PostgreSQL repose sur plusieurs critères clés : sa fiabilité, sa scalabilité et sa maturité en production. En plus de gérer des données relationnelles

classiques (logs, utilisateurs, paramètres...), il peut aussi indexer efficacement des vecteurs issus des modèles d'embedding, et permettre des requêtes de type nearest neighbor search.

Son intégration avec Spring Boot est fluide grâce à JPA ou JDBC, et son usage dans ce projet est évolutif : dans un premier temps, les embeddings sont stockés en mémoire (InMemoryEmbeddingStore), mais la bascule vers PostgreSQL permettra une persistance et une scalabilité bien supérieures, tout en conservant une interface compatible avec les composants LangChain4j.

Ainsi, PostgreSQL joue un rôle fondamental dans la stratégie d'enrichissement contextuel des requêtes utilisateur, en permettant à l'IA de s'appuyer sur des documents pertinents stockés localement de façon fiable et interrogeable.

La figure 3.1 illustre de manière synthétique les interactions entre les principaux composants techniques de l'architecture mise en place dans le cadre de ce projet.

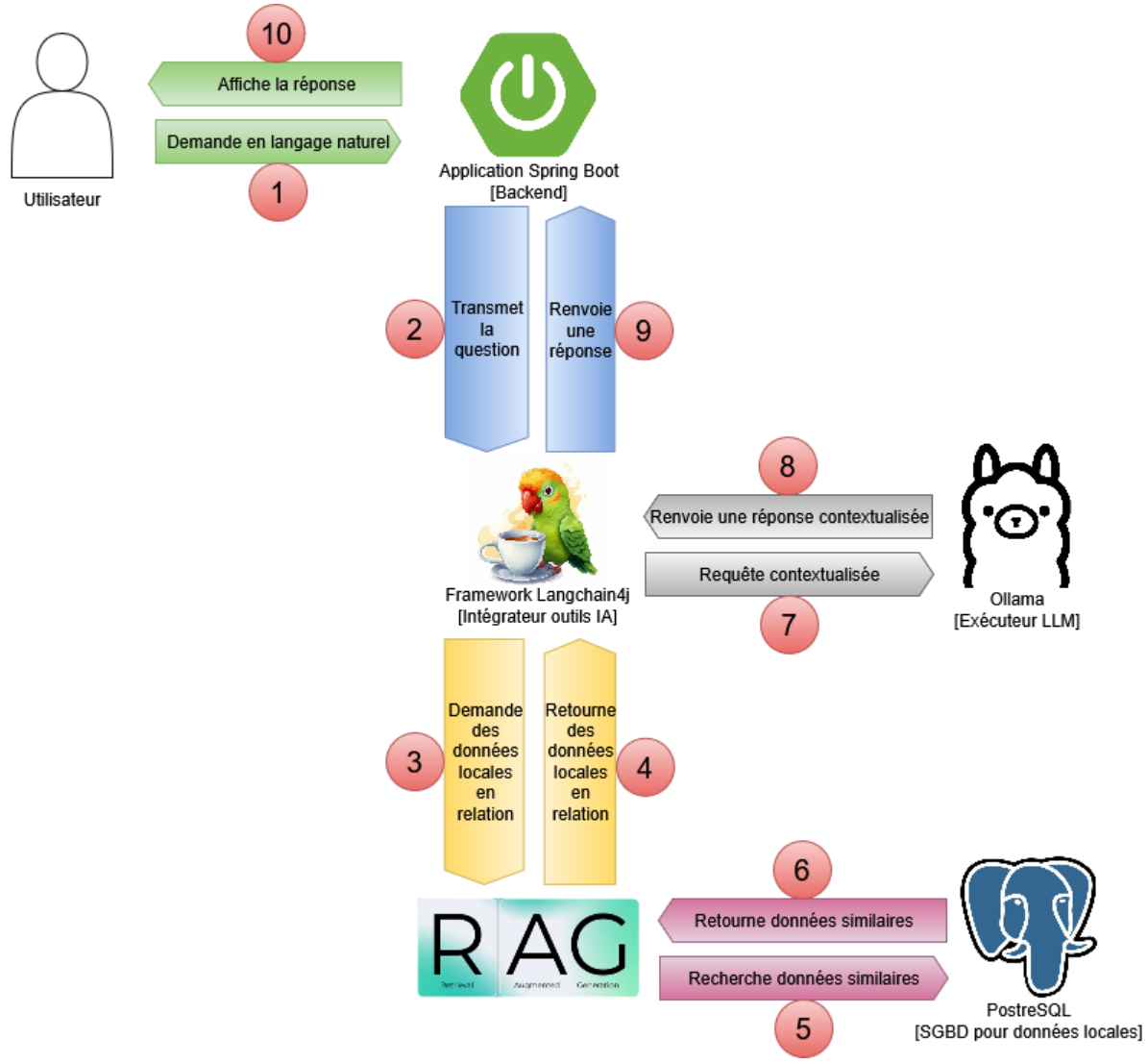


FIGURE 3.1 – *Diagramme d'architecture technique*

3.3 Bonnes pratiques appliquées du développement Java

Dans le développement logiciel, l'adoption de bonnes pratiques de codage est essentielle pour garantir la robustesse, la maintenabilité et l'évolutivité des applications. Un code bien structuré, avec une logique claire et une réduction des redondances, facilite non seulement les futures modifications, mais améliore aussi la collaboration entre développeurs. Cette section aborde des principes clés pour optimiser l'écriture du code, en mettant l'accent sur des méthodes qui en améliorent la qualité tout en réduisant les risques d'erreurs.

3.3.1 Utilisation de Bibliothèques Matures : Apache Commons

Plutôt que de réinventer la roue, il est recommandé d'utiliser des bibliothèques robustes comme Apache Commons, qui fournit des utilitaires optimisés pour plusieurs opérations telles que la manipulation de collections, les opérations sur les chaînes, et les validations.

Pour mettre en lumière les utilisations de ces utilitaires dans notre projet, prenons par exemple la méthode `estimateTokenCountInText(String text)` de la classe `TokenizerMyAppImpl`, dont la figure 3.2 montre une implémentation classique.

```
@Override  12 usages
public int estimateTokenCountInText(String text) {
    if (text == null || text.isBlank())
        return 0;
    return tokenizer.encode(text).getTokens().length;
}
```

FIGURE 3.2 – *Implémentation sans utiliser Apache Commons*

Cette approche, bien que fonctionnelle, présente plusieurs limitations : la combinaison de deux vérifications dans une même condition, la duplication de cette logique à de nombreux endroits du code, et le risque potentiel de `NullPointerException`.

L'adoption d'une nouvelle implémentation exploitant la bibliothèque Apache Commons est montrée dans la figure 3.3.

```
@Override  12 usages
public int estimateTokenCountInText(String text) {
    if (StringUtils.isBlank(text))
        return 0;
    return tokenizer.encode(text).getTokens().length;
}
```

FIGURE 3.3 – *Implémentation en utilisant Apache Commons*

La méthode isBlank() de la classe StringUtils retourne true dans trois cas : si text est null, ou une chaine de caractères vides, ou une chaine de caractères ne contenant que des espaces blancs, Ce qui donne plus de robustesse au projet, en évitant les NullPointerException sans besoin de vérifier null explicitement, améliore la lisibilité en remplaçant une condition complexe par un seul appel clair, et garantit un comportement uniforme dans tout le projet.

3.3.2 Approches de parcours de collections en Java

Durant le développement, j'ai été amené à manipuler des collections de données. Plusieurs approches s'offraient alors à moi, chacune présentant des caractéristiques distinctes :

- **La boucle for traditionnelle** : Approche historique de Java, elle se base sur un index numérique. Bien que simple conceptuellement, elle montre plusieurs limitations : une syntaxe verbeuse nécessitant la déclaration d'un compteur, un risque d'erreurs d'indice (`IndexOutOfBoundsException`), une inadaptation aux structures non indexées comme les Set, et une difficulté à maintenir pour des traitements complexes.
- **La boucle for-each** : Introduite dans Java 5, elle simplifie le parcours, en offrant une syntaxe plus concise et plus lisible, et en éliminant le risque d'erreurs d'indice, cependant elle ne permet pas de modification concurrente, en plus d'être limitée à un parcours séquentiel simple.
- **L'Iterator** : Fournit un contrôle fin sur le parcours, en effet, il permet la suppression d'éléments (méthode `remove()`), et fournit une interface standardisée pour toutes les collections. Par contre, il requiert une gestion manuelle fastidieuse, et le code produit est peu lisible pour des opérations complexes.
- **Les Streams (approche retenue)** : Les Streams sont une abstraction introduite avec Java 8 qui permettent de traiter des collections de données de façon fonctionnelle, fluide et lisible. Un Stream représente une séquence d'éléments que l'on peut traiter (filtrer, transformer, agréger, etc.) sans modifier la source d'origine (par exemple, une List ou un Set). Cette approche moderne présente des avantages déterminants : une syntaxe déclarative exprimant l'intention métier, un chaînage des opérations, une possibilité de parallélisation transparente, et une meilleure maintenabilité du code.
- **ParallelStream** : Extension des Streams standard introduite avec Java 8, elle permet un traitement parallèle automatisé des collections en tirant parti des architectures multi-cœurs. Contrairement aux Streams séquentiels, ParallelStream partitionne les données en sous-ensembles traités simultanément par différents threads (via le Fork/Join Pool). Ses avantages incluent une accélération des traitements pour les opérations CPU-intensives tels que les traitements de gros volumes de données, une syntaxe identique aux Streams, et une abstraction de la complexité, en effet

la gestion du multithreading est effectuée sans manipulation manuelle de threads. Le tableau 3.1 expose une comparaison entre ces approches selon différents critères.

TABLE 3.1 – Comparaison des méthodes de parcours en Java

Critère	`for`	`for-each`	`Iterator`	`Stream`	`ParallelStream`
Modification possible	Risqué	Interdit	`remove()`	Interdit	Interdit
Accès par index	Oui	Non	Non	Non	Non
Lisibilité	Moyenne	Bonne	Faible	Excellente (déclaratif)	Excellente (déclaratif)
Types de sources	List/ Array	`Iterable`	`Collection`	Collection, Array, flux, etc.	Collection, Array, flux, etc.
Performance	Optimale	Légère baisse	Légère baisse	Bonne (selon cas)	Variable (multi-thread)
Opérations intégrées	Non	Non	Non	Oui	Oui
Parallélisme	Difficile	Impossible	Impossible	Possible	Automatique (Fork/Join)
Gestion du `null`	Manuel	Manuel	Manuel	`Optional`	`Optional`
Cas d'usage typique	Parcours indexé	Parcours simple séquentiel	Suppression d'éléments	Traitement fonctionnel	Traitement parallèle intensif

Prenons un exemple de notre projet, la méthode estimateTokenCountInTools(Iterable<Object> objectsWithTools) de la classe TokenizerMyAppImpl a été implémentée dans un premier temps en utilisant une boucle foreach, comme le montre la figure 3.4.

```
@Override  no usages
public int estimateTokenCountInTools(Iterable<Object> objectsWithTools) {
    int total = 0;
    if (objectsWithTools != null) {
        for (Object obj : objectsWithTools) {
            total += estimateTokenCountInTools(obj);
        }
    }
    return total;
}
```

FIGURE 3.4 – *Implémentation sans utiliser les Streams*

Une évolution de cette implémentation utilisant les Streams ainsi que la bibliothèque Apache Commons est illustrée dans la figure 3.5.

```
@Override  no usages  hicham-loudiyi
public int estimateTokenCountInTools(Iterable<Object> objectsWithTools) {
    return StreamSupport.stream(
                    IterableUtils.emptyIfNull(objectsWithTools).spliterator(),
                    parallel: true) Stream<Object>
            .mapToInt(this::estimateTokenCountInTools) IntStream
            .sum();
}
```

FIGURE 3.5 – *Implémentation en utilisant les Streams*

Cette implémentation, offrant une gestion robuste des null avec `IterableUtils.emptyIfNull()`, convertit l'Iterable en Stream parallélisable avec `StreamSupport.stream(..., true)`, ce qui permet un traitement réparti sur plusieurs cœurs CPU si la collection est grande, et une optimisation automatique pour les gros volumes de données, et finalement effectue un chaînage clair des opérations en appliquant les méthodes `mapToInt(...)` et `sum()`.

3.3.3 Importance de la factorisation du code

La factorisation du code consiste à regrouper dans des méthodes ou classes dédiées les portions de logique qui se répètent à plusieurs endroits du programme. Cette pratique s'inscrit dans les bonnes pratiques du développement logiciel, notamment le principe *DRY* (*Don't Repeat Yourself*), qui préconise d'éviter les duplications de code.

Factoriser améliore la lisibilité, la maintenabilité, et réduit les risques d'erreurs en centralisant les modifications à un seul endroit. Cela permet également de clarifier les responsabilités des différentes classes, en accord avec le principe de responsabilité unique *SRP* (*Single Responsibility Principle*) du modèle SOLID.

La figure 3.6 montre une portion de code répétée avant factorisation.

```java
@Override  no usages  hicham-loudiyi
public int estimateTokenCountInToolSpecifications(Iterable<ToolSpecification> toolSpecifications) {
    return StreamSupport.stream(
                    IterableUtils.emptyIfNull(toolSpecifications).spliterator(),
                    parallel: true) Stream<ToolSpecification>
            .mapToInt( ToolSpecification spec -> estimateTokenCountInText(spec.toString())) IntStream
            .sum();
}

@Override  no usages  hicham-loudiyi
public int estimateTokenCountInToolExecutionRequests(Iterable<ToolExecutionRequest> toolExecutionRequests) {
    return StreamSupport.stream(
                    IterableUtils.emptyIfNull(toolExecutionRequests).spliterator(),
                    parallel: true) Stream<ToolExecutionRequest>
            .mapToInt( ToolExecutionRequest req -> estimateTokenCountInText(req.toString())) IntStream
            .sum();
}
```

FIGURE 3.6 – *Exemple de code répétitif avant factorisation*

Dans la figure 3.6, chacune des méthodes `estimateTokenCountInToolSpecifications` et `estimateTokenCountInToolExecutionRequests` permet d'utiliser un Stream pour estimer un nombre de tokens sur un itérable, c'est donc une logique répétée, seuls le type d'objets de l'Iterable et la méthode de comptage appliquée à chaque objet sont différents. La figure 3.7 illustre la version améliorée après factorisation.

```java
private <T> int countTokensInIterable(Iterable<T> iterable, ToIntFunction<T> tokenCounter) {  3 usages  hicham-loudiyi
    return StreamSupport.stream(
                    IterableUtils.emptyIfNull(iterable).spliterator(),
                    parallel: true) Stream<T>
            .mapToInt(tokenCounter) IntStream
            .sum();
}

@Override  no usages  new *
public int estimateTokenCountInToolSpecifications(Iterable<ToolSpecification> toolSpecifications) {
    return countTokensInIterable(toolSpecifications,  ToolSpecification spec -> estimateTokenCountInText(spec.toString()));
}

@Override  no usages  new *
public int estimateTokenCountInToolExecutionRequests(Iterable<ToolExecutionRequest> toolExecutionRequests) {
    return countTokensInIterable(toolExecutionRequests,  ToolExecutionRequest req -> estimateTokenCountInText(req.toString()));
}
```

FIGURE 3.7 – *Code refactorisé avec méthode utilitaire*

La logique répétée est regroupée dans la méthode utilitaire `countTokensInIterable(Iterable<T> iterable, ToIntFunction<T> tokenCounter)`, puis réutilisée en appelant cette méthode au besoin.

On constate une nette amélioration de la clarté du code, les règles de validation sont centralisées dans une méthode dédiée, facilitant ainsi leur réutilisation et leur évolution future.

3.3.4 Utilisation d'une classe de configuration

L'intégration d'une classe de configuration dans ce projet est cruciale pour standardiser et optimiser la gestion des paramètres techniques et métiers. Par exemple, des valeurs comme le nom du modèle de language, sa température, le maximum de messages du ChatMemory, des paramètres du Retriever, la taille maximale des images uploadées et bien d'autres, ont été externalisées dans une classe dédiée que nous avons nommée `ConfigurationPropertyValue` et injectées - soit depuis un fichier de configuration comme `application.properties`, ou bien depuis une base de données - via l'annotation `@Value`. De plus, cette dernière offre la possibilité de définir des valeurs par défaut en cas d'absence de valeurs à injecter.

La figure 3.8 montre un morceau de la classe `ConfigurationPropertyValue`.

```
@Component  7 usages  hicham-loudiyi *
@Getter
public class ConfigurationPropertyValue {

    @Value("${max.imagesize:5242880}")
    private long maxImageSize;

    @Value("${ollama.baseurl:http://localhost:11434}")
    private String ollamaBaseUrl;

    @Value("${ollama.modelname:qwen2.5vl:7b}")
    private String ollamaModelName;

    @Value("${ollama.temperature:0.1}")
    private double temperature;

    @Value("${ollama.timeout:5}")
    private int ollamaTimeout;

    @Value("${ollama.chatmemory.maxmessages:20}")
    private int chatMemoryMaxMessages;

    @Value("${ollama.embeddings.modelname:nomic-embed-text}")
    private String embeddingsModelName;
```

FIGURE 3.8 – *Un morceau de la classe de configuration*

Pour récupérer ces valeurs dans une autre classe, il suffit d'y injecter un bean de la classe de configuration, comme le montre l'exemple dans la figure 3.9.

```java
@Configuration  hicham-loudiyi
@Slf4j
@RequiredArgsConstructor
public class AiConfig {

    private final ConfigurationPropertyValue config;

    @Bean  hicham-loudiyi
    public ChatLanguageModel llm() {
        return OllamaChatModel.builder()
                .baseUrl(config.getOllamaBaseUrl())
                .modelName(config.getOllamaModelName())
                .temperature(config.getTemperature())
                .timeout(Duration.ofMinutes(config.getOllamaTimeout()))
                .build();
    }
```

FIGURE 3.9 – *Utilisation de la classe de configuration*

Avec cette approche on peut ajuster les paramètres techniques sans toucher au code métier et sans recompilation, on gagne plus de flexibilité en adaptant les paramètres à l'environnement (dev, test, prod) via des profils Spring, et on respecte l'un des principe SOLID, qui est la séparation des responsabilités, puisque le service se concentre sur la logique métier, tandis que la configuration technique est externalisée.

3.4 Captures d'écran

3.4.1 Premier prototype

Pour atteindre nos objectifs ambitieux, nous avons suivi une approche incrémentielle. Dans un premier temps, un prototype fonctionnel a été réalisé afin de valider les fondements techniques du projet. Ce prototype est une simple application basée sur un agent intelligent exploitant une architecture RAG unimodale, capable de répondre aux questions de l'utilisateur à partir d'un document texte ou PDF fourni. Cette première version a permis de comprendre le fonctionnement du framework LangChain4j, de tester l'intégration avec Ollama, et de valider le concept de récupération de contexte à partir de documents externes.

Pour tester ce prototype, nous avons fourni un fichier PDF, son contenu est une lettre de recommandation pour une étudiante appelée Nour, nous avons ensuite posé une question à propos de cette étudiante à l'agent AI, en utilisant un contrôleur web. La figure 3.10 montre le résultat obtenu.

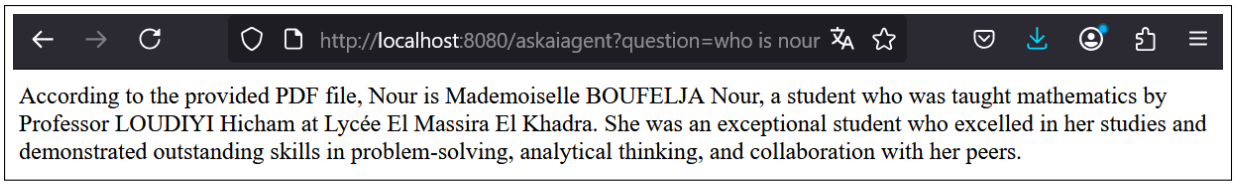


FIGURE 3.10 – *Test du premier prototype*

Le modèle choisi pour tester ce prototype est Llama3, le prototype à réussi à répondre correctement à la question posée.

3.4.2 Dernière version

Structure du projet Java

Une bonne structuration d'un projet logiciel constitue un fondement essentiel pour assurer sa lisibilité, sa maintenabilité et son évolutivité. Dans notre cas, le projet respecte les conventions standard de l'écosystème Java et de l'architecture Spring Boot. L'arborescence suit une séparation claire entre les différentes couches de l'application, notamment les agents (`agents`), la configuration (`config`), les objets de transfert de données (`dto`), les exceptions personnalisées (`exceptions`), les services métier (`service`) et la couche de présentation via les contrôleurs web (`web`). Cette organisation modulaire favorise l'encapsulation des responsabilités, en conformité avec les principes SOLID, et facilite les tests unitaires ainsi que l'intégration continue. Par ailleurs, l'intégration des ressources statiques et de configuration dans les répertoires `resources/static` et `application.properties` respecte les conventions de Spring Boot, permettant un déploiement harmonisé.

La capture d'écran dans la figure 3.11 illustre la structure de notre projet Java.

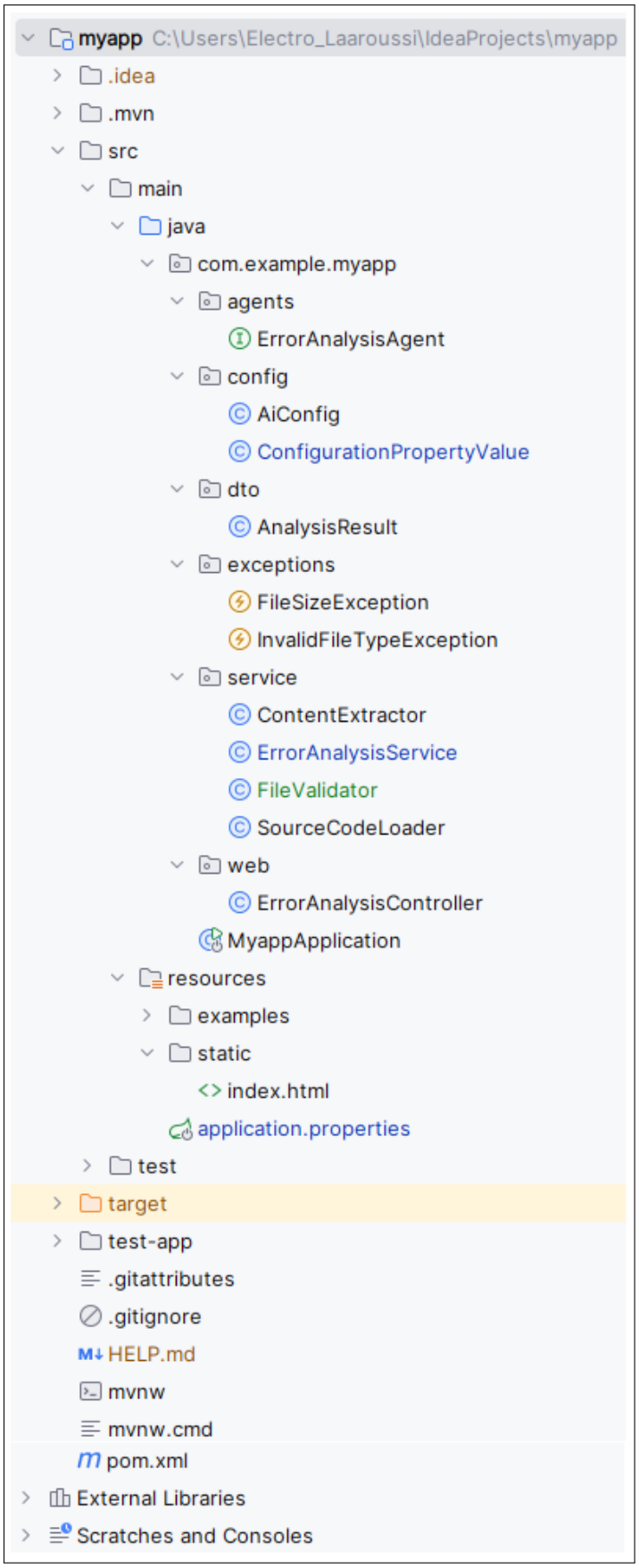


FIGURE 3.11 – *Structure du projet Java*

Configuration des modèles et chargement des ressources pour le RAG

Afin de permettre à notre application de traiter des entrées de nature hétérogène (texte et image), le recours à un modèle de langage multimodal s'est révélé indispensable. À cet effet, le modèle `Qwen2.5vl:7b` a été choisi pour ses capacités avancées d'analyse conjointe du contenu textuel et visuel.

Plusieurs autres paramètres ont été configurées, telles que les paramètres relatifs au LLM, au modèle d'Embeddings, au Retriever, et la taille maximale des images uploadées.

D'autre part, le service `sourceCodeLoader` est configuré à charger les fichiers du code source de l'application présentant des erreurs, en lui disposant de son chemin. Ces fichiers constituent la matière première pour le RAG, qui les utilisera pour enrichir sa base de connaissances, afin d'obtenir des réponses cohérentes avec le contexte de l'application avec des erreurs.

La figure 3.12 montre une partie du fichier de configuration `application.properties`.

```
max.imagesize=5242880

ollama.baseurl=http://localhost:11434
ollama.modelname=qwen2.5vl:7b
ollama.temperature=0.1
ollama.timeout=120

ollama.chatmemory.maxmessages=20

ollama.embeddings.modelname=nomic-embed-text
ollama.embeddings.timeout=5

contentretriever.maxresults=2
contentretriever.minscore=0.6

documentsplitter.maxsegmentsizeintokens=1000
documentsplitter.maxoverlapsizeintokens=100

sourcecode.path=C:\\Users\\Electro_Laaroussi\\IdeaProjects\\TestAppWeb\\src

api.paths.errors=/api/errors
```

FIGURE 3.12 – *Morceau du contenu du fichier application.properties*

Démarrage de l'Application Spring Boot

Au démarrage de l'application, Spring Boot initialise le contexte d'exécution en suivant une séquence bien définie. Tout commence par le chargement de la classe principale `MyappApplication`, annotée avec `@SpringBootApplication`, qui active la configuration automatique, la scan des composants et les propriétés définies dans `application.properties`. Les beans déclarés dans `AiConfig` sont instanciés, configurant notamment le modèle

RAG et les services dépendants comme `ErrorAnalysisAgent` ou `SourceCodeLoader`. En parallèle, Spring MVC s'initialise pour préparer les endpoints du contrôleur `ErrorAnalysisController`. Enfin, le serveur embarqué Tomcat démarre et expose l'API sur le port configuré, rendant accessible l'interface Swagger UI pour tester les endpoints. Cette orchestration garantit que tous les services et composants sont prêts à traiter les requêtes dès le démarrage complet.

Conformité aux Normes RESTful et Bonnes Pratiques d'API

Notre projet respecte les principes fondamentaux d'une API RESTful. En effet L'endpoint `/api/errors` suit une sémantique HTTP claire, la méthode POST est utilisée pour la création d'une ressource (analyse d'erreur), conformément aux verbes REST, et les réponses HTTP sont standardisées (200 pour le succès, 400/413/415/500 pour les erreurs métier), avec des codes de statut pertinents et des messages structurés dans le DTO `AnalysisResult`. L'utilisation de `@RestController` et de `MediaType.MULTIPART_FORM_DATA_VALUE` garantit une sérialisation/désérialisation transparente des données.

De plus, l'API développée est documentée via Swagger (annotations `@Tag`, `@Operation`), offrant une description claire des fonctionnalités et des schémas de réponse. L'acceptation de données binaires (captures d'écran) via MultipartFile et leur validation (taille et type) illustrent également la prise en compte des contraintes RESTful sur les formats de données.

Création d'une page web pour effectuer des tests

Nous avons conçu une interface web minimaliste permettant de soumettre les erreurs rencontrées dans les applications. Cette page offre un formulaire où l'utilisateur peut poster une description détaillée - et/ou une stacktrace - du problème ainsi qu'une capture d'écran optionnelle illustrant l'anomalie, un bouton "Analyser l'erreur", et un champs de texte où la réponse générée va être affichée.

La figure 3.13 affiche une capture écran de cette interface.

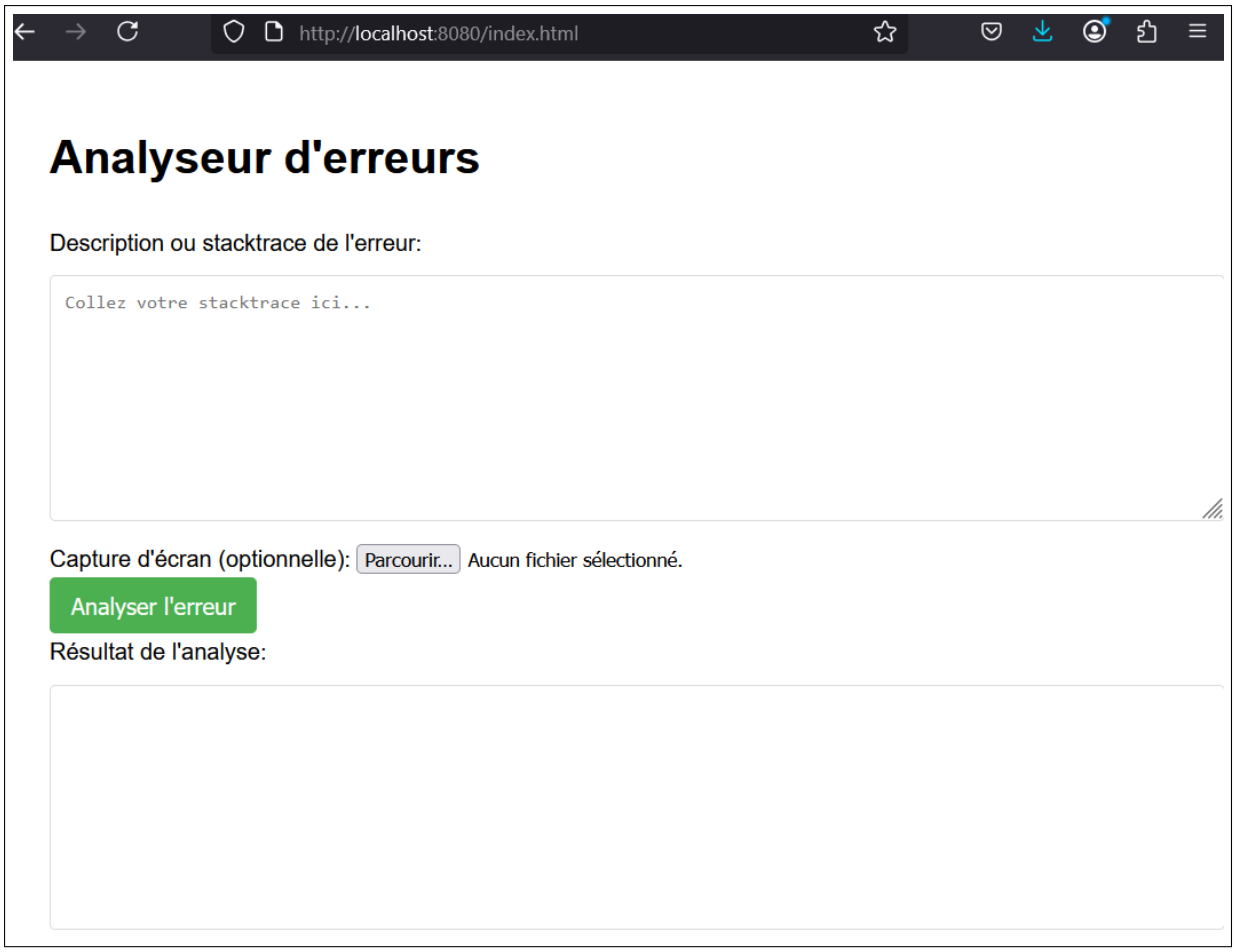


FIGURE 3.13 – *Interface de l'application de test*

Simulation d'une application qui génère une erreur

Afin de simuler un cas réel d'erreur dans une application, nous avons développé une application web de test générant intentionnellement une `NullPointerException`. Cette exception, fréquente dans le développement logiciel, se produit lorsqu'une méthode tente d'accéder à un objet non initialisé (`null`).

La figure 3.14 montre le résultat d'exécution sur un navigateur.

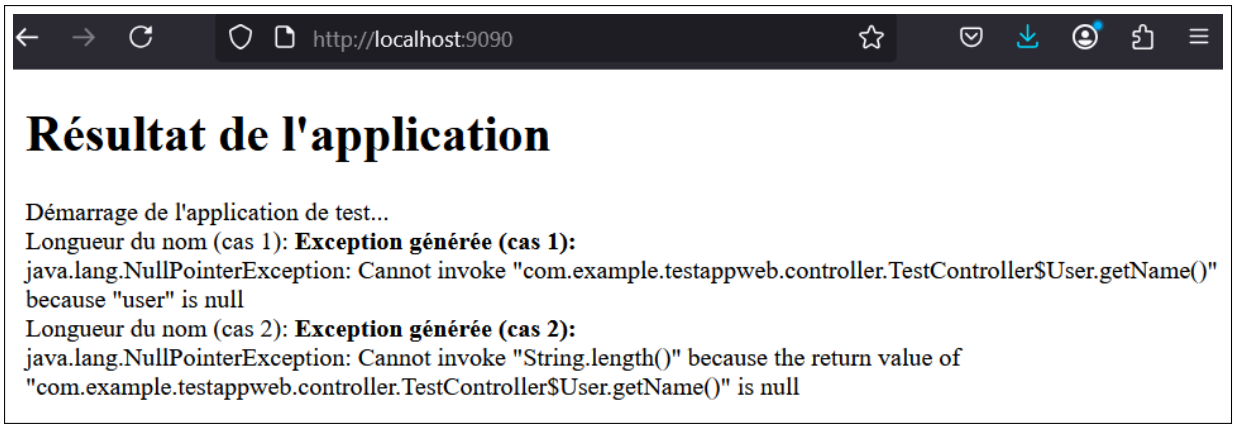


FIGURE 3.14 – *Résultat d'exécution de l'application de test*

Premier test

L'objectif était de valider la capacité de notre application intelligente à détecter automatiquement cette anomalie, à en analyser les causes (comme une variable non instanciée ou un retour de méthode null), et à proposer un correctif approprié, tel qu'une vérification de nullité ou une initialisation par défaut.

Ce test a permis de confirmer l'efficacité de notre système dans la gestion des erreurs courantes.

Conclusion

Annexe

Webographie