Rapport de projet

Mail Stone

Groupe 4

Idriss BENGUEZZOU
Ali BOUGASSAA
Ghilas MEZIANE
Mohammed ROUABAH

21/11/2022

Table des matières

1 Introduction

L'objectif de ce projet est de simuler la communication entre différentes applications et ce par l'intermédiaire de messages structurés. Ce système de communication peut être utilisé entre les entreprises pour échanger avec leurs clients, fournisseurs ou partenaires. Ainsi, ce projet a pour but de traiter l'extraction de données à partir de textes, la conception et l'alimentation d'une base de donnée à l'aide de données issues de l'analyse, la génération de contenus structurés et bien formés dédiés aux applications métiers. Pour concrétiser ce projet nous avons formé un quadrinôme, constitué de BENGUEZZOU Idriss, MEZIANE Ghilas, BOUGASSAA Ali et ROUABAH Mohammed.

2 Gestion de projet

Afin de faciliter la gestion de ce projet nous avons utilisé TRELLO en y appliquant la méthode Kanban. Cet outil nous a permis de lister les tâches au fur et à mesure et à les répartir équitablement tout en prenant en compte les compétences de chacun. Nous avons également utilisé un logiciel de gestion de versions décentralisé (Gitlab), afin d'avoir le projet à jour sur toutes nos machines.

3 Rappel... MailStone, quésaco ?

Les documents traités par MailStone sont les mails de réclamation adressés à un support client - type SAV – pour une entreprise proposant divers services (matériels ou digitaux). Les différentes parties exploitables que nous avions listé sont les suivantes :

— **Le souci :** la panne, le problème, la préoccupation du client, un défaut de prestation, une demande d'aide...

— **Le produit :** le type de produit, la référence, le nom du produit, savoir s'il est garantie ou pas...

— **Les dates :** date de l'envoi de la réclamation, date de la panne, date de traitement de la demande, date de résolution du problème, date d'achat du produit...

— **Les coûts :** liées au produit, à sa réparation, la main d'œuvre, les potentielles pièces de rechanges...

— **Les coordonnées** des clients (soit pour fidéliser, soit pour détecter les comportements abusifs).

— **Les solutions :** la ou les réponses apportées au client, l'intervention ou pas de techniciens, si le produit a été échangé ou remboursé...

4 Technologies utilisées

Pour mener à bien ce projet, nous avons choisi d'utiliser **Spring Boot**, un framework **java** qui facilite le développement d'applications fondées sur Spring. Ce choix a été motivé par notre volonté d'approfondir les connaissances acquises dans cette technologie, grâce à l'UE programmation web avancée.
Spring Boot ne suffit pas à lui même pour créer une application web complète, il était donc nécessaire d'utiliser des technologies orientées frontend tel que le langage **HTML** et **CSS** pour la création d'une interface ergonomique, ou encore javascript pour la manipulation du DOM.

5 Architecture du projet

Le projet est composé de quatre applications : client, mail, analyseur et serveur. Nous détaillerons plus bas le fonctionnement de celles-ci. Chacune de ces applications respecte un patron de conception bien défini. L'application métier (client) ainsi que l'application mail et le serveur respectent le motif d'architecture MVC (Modèle-vue-contrôlleur). Le serveur réalise des requêtes sur la base de données c'est pourquoi nous avons décidé d'utiliser le modèle DAO pour organiser la gestion des accès à la BDD.

5.1 La base de données

La base de données final du projet est la suivante :

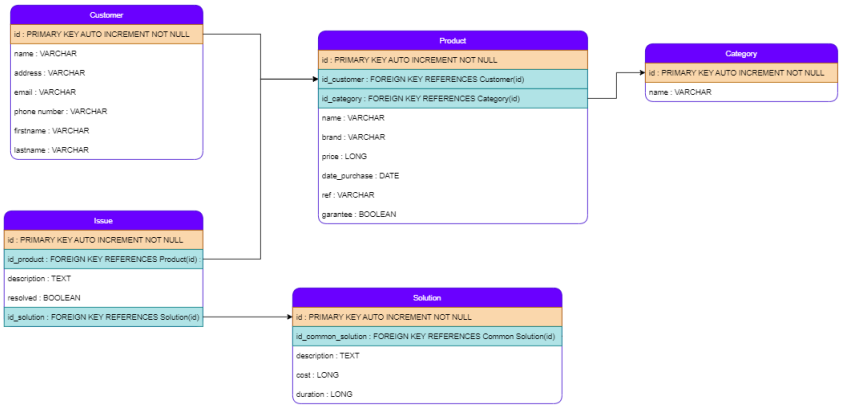

FIGURE 1 – Schéma de la base de données

6 Comment fonctionne MailStone

MailStone est un projet simulant la communication entre une ou plusieurs application métier.

Pour ce faire nous avons décidé de créer une boite mail commune qui est censée recevoir les mails de réclamations adressées a l'entreprise. Les mails pourront ainsi être envoyés à cette adresse depuis une application web possédant une interface permettant la rédaction d'un mail. Cette boite mail commune est également reliée à une application tierce que nous avons nommée "analyzer", qui se chargera de récupérer les nouveaux mails, les analyser et transmettre les

données extraites à notre "server" qui s'occupera quant à lui d'alimenter la base de données. Notre application métier, nommé "client", offre la possibilité aux utilisateurs de réaliser des recherches en questionant le serveur et d'exploiter les données collectées sous différentes formes.

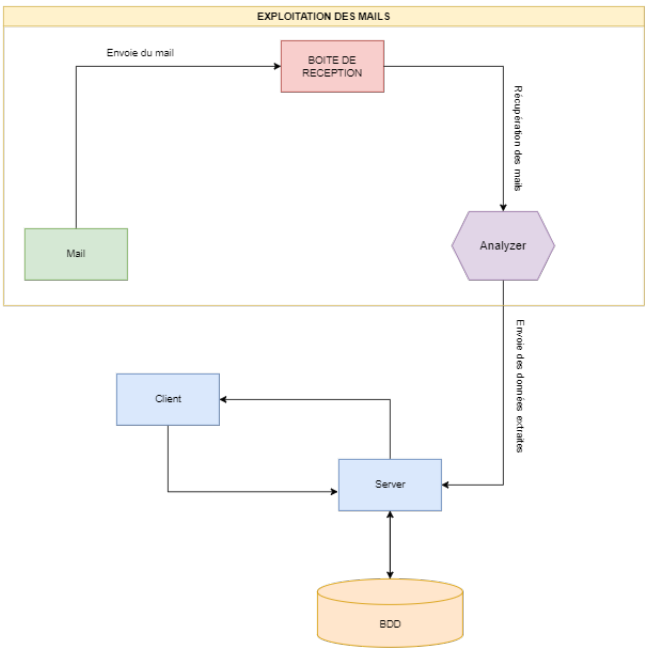

FIGURE 2 – Une vue d'ensemble des applications

7 Explication approfondie

7.1 L'exploitation des mails

7.1.1 Récupération des mails

Nous avons crée une boîte mail ayant pour adresse "mailstone2022@gmail.com" destinée à recevoir les réclamations des clients.

Dans un premiers temps, nous avons voulu récuperer les mails reçus sur notre boîte GMAIL. Pour cela nous avons utilisé l'API "Javax.Mail". Nous nous connectons à notre boite de réception directement à partir de notre analyseur en utilisant le protocoloe pop3 et un mot de passe personnalisé pour notre application fourni par Google permettant de sécuriser l'accès via une application tierce. A l'aide de cette API, nous avons configuré quel dossier devrait être scruté par notre application (pour nous ce dossier sera le dossier principal de la boite de récepetion, le dossier "Inbox"), pour récuperer son contenu.

Une contrainte s'est alors posée lors de l'analyse des mails envoyés directement via gmail par l'utilisateur, en effet pour analyser le corps du mail, il est nécessaire d'avoir un format spécial. Comment pouvons-nous être sûr que l'utilisateur respecte ce format alors qu'il écrit de sa propre boîte mail ?
La solution mise en place, a été la réalisation d'un formulaire permettant l'envoi de mail sous le bon format et ceci afin de guider l'utilisateur d'une part, et de nous faciliter la tâche d'autre part. Nous avons tenu à laisser un maximum de liberté à l'utilisateur en ne mettant que deux

champs dans le formulaire. Le premier permettant à l'utilisateur de décrire le produit, et le second permettant à l'utilisateur de décrire la panne. Ce choix nous a contraint à rechercher des méthodes d'analyse de texte pour nous permettre d'extraire les données qui nous intéressent.

7.1.2 L'analyse

Maintenant que nous avons accès au contenu de notre boite de réception, nous pouvons nous intéresser à la manière la plus efficace pour traiter le texte. Dans un premier temps, il a fallu formatter le corps du mail afin de nous débarasser des caractères spéciaux, des retours à la ligne, des espaces en trop...

Une fois le texte brut nettoyé, il fallait trouver des délimiteurs dans le texte permettant d'extraitre les données qui nous intéressent. Ces délimiteurs sont des mots clés tels que "marque, référence, ref...". Cependant, il arrive que l'utilisateur commette des erreurs d'orthographe ce qui nous à amener à mettre en place une distance de Levenshtein.

Cette distance appliquée à nos mots clés prédéfinis, nous permet d'extraire les données dont nous avons besoin. Par exemple, lorsque l'utilisateur écrit "marquz LG" notre algorithme permet de savoir que cela correspond au mot clé "marque" et le mot suivant "LG" nous permets de remplir le champ "marque" de l'objet produit.

7.2 Server

Le "server" est l'application permettant de réaliser les requêtes sur la base de données, c'est elle qui va interpréter les messages envoyés depuis les applications tierces et leur répondre, dans un langage commun. Le langage choisi pour réaliser tout échange entre les applications est l'Extensible Markup Language (XML), qui est un langage de balisage permettant de structurer les données.

7.2.1 Lecture des questions

Chaque question est identifiable par un code qui lui est propre. L'application server récupère ce code lors de la lecture de la question et, l'utilise dans le but de convertir cette question en requête SQL, qu'elle adressera à la base de données. Pour permettre la conversion XML->SQL, nous avons développé une classe nommée *XMLReader.java* qui implémente des méthodes permettant la lecture de balises XML. De plus, nous avons implémenté dans cette même classe des méthodes se servant de *XPATH* pour localiser et récupérer les valeurs recherchées dans le document XML.

7.2.2 Génération des réponses

Une fois la question lue, interprétée, transformée en SQL, le serveur va s'ateller à générer une réponse en se servant de données qu'il va collecter directement dans la base de données à l'aide des class DAO. De la même manière que les questions, les réponses possèdent également un code qui permet de les identifier. Nous avions dans un premier temps créé un format de fichiers XML pour chaque réponse. Cependant, nous avons remarqué une redondance dans le code qui pouvait être évitée en unifiant les réponses sous un seul et même format.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE issues SYSTEM "communication\DTD\response\response.dtd">
<issues code="r001">
    <issue>
        <product>
           ...
            <category>
             ...
            </category>
            <client>
                ...
            </client>
        </product>
        <solution>
            ...
        </solution>
    </issue>
</issues>
```

L'écriture se fait grace à notre classe *XMLWriter.java* qui implémente les méthodes permettant l'écriture du contenu du fichier. Nous avons également, pu implémenter une methode permettant le formattage du contenu du fichier XML, nous permettant d'obtenir un fichier bien indeté et structuré.

7.3 L'application métier

L'application métier offre aux utilisateurs la possibilité d'accéder à une interface permettant de poser des questions par le biais de formulaires. Elle va ensuite se charger de les structurer en convertissant les entrées de l'utilisateur au format XML.

7.3.1 Création de questions

Nous utilisons la même logique que pour l'application "server" à quelques détails près. En effet, la génération des questions dépend encore une fois du le code de la question. Néanmoins, à l'instar de l'application "server" qui se sert de la BDD, les données proviennent des entrées de l'utilisateur.

7.3.2 Lecture de la réponse

La réponse possède un code commun a toutes les réponses. Il est nécessaire pour pouvoir présenter la réponse à l'utilisateur de lire et de transformer le fichier XML en POJO (Plain Old Java Object). Pour cela, nous avons utilisé l'interface Unmarshaller qui permet, à l'aide des annotations XML existantes à cet effet, d'injecter dans nos model les données du fichiers XML.

7.3.3 Affichage des données

Une fois les données extraites dans nos objets java, il nous faut les afficher à l'utilisateur, pour cela nous avons pu créer des pages HTML permettant de mettre en forme les données receuillies.

L'utilisateur aura donc ainsi un rendu accessible sur son écran.

8 Communication

Pour qu'un message soit lu par l'application "server", il faut le lui transmettre. Pour établir cette communication nous avons utilisé deux méthodes. La première permet de simuler l'envoi de fichiers tandis que la seconde permet d'envoyer le fichier en se servant de protocoles.

8.1 Une simulation

Pour simuler l'envoi de fichiers d'une application à une autre, nous avons mis en place un dossier appelé "communication", celui-ci va nous permettre de sauvegarder les messages que les applications s'échangent. Ainsi, l'application "server" et l'application métier pourront communiquer par le biais de ce dossier. Le dossier est organisé de la manière suivante :

FIGURE 3 – Architecture du dossier de communication

Nous pourrons retrouver dans le dossier nommé "process", les questions et réponses en cours de traitement, dans le dossier "archived" nous retrouverons les messages traités et archivés, celui-ci nous permet de garder une trace des échanges réalisés entre l'application "server" et des applications tierces. Enfin, le dossier DTD contient toutes les dtd écrites pour valider les messages XML.

Cependant, pour établir une communication dynamique il est nécessaire de détecter les nouvelles questions pour que l'application "server" puisse la traiter, et de détecter les nouvelles réponses pour que l'application métier puisse les lire et afficher le résultat à l'utilisateur. C'est pourquoi, nous avons crée une class *Watcher.java*, permettant de scruter un dossier et de notifier un controller de l'arrivée d'un nouveau message.

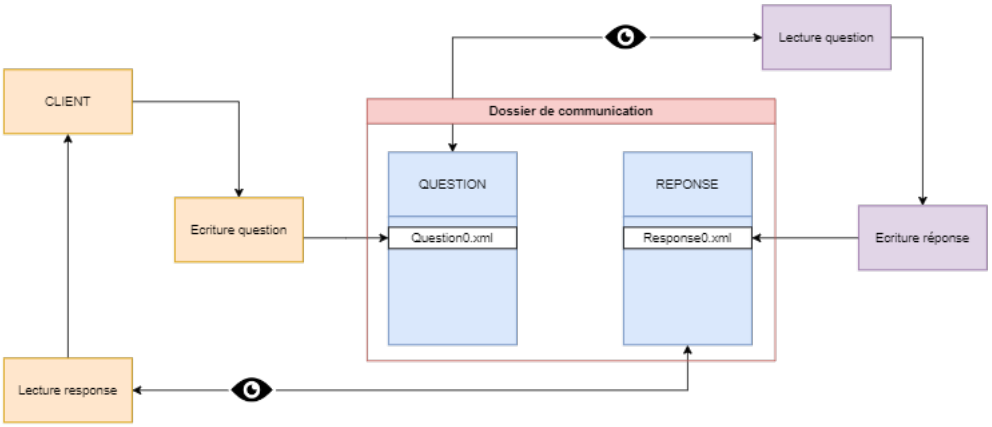

FIGURE 4 – Schéma de communication

8.2 Requête Http

Nous avons voulu apprendre à réaliser une "vraie" communication entre applications à l'aide de protocoles HTTP, c'est pourquoi nous avons pu implémenter une class permettant la réalisation de requête HTTP. Nous avons utilisé la méthode HTTP POST qui permet d'envoyer des données au serveur. Le type du corps de la requête est indiqué par l'en-tête Content-Type, qui pour nous correspondait à l'entête "application/xml" puisque nos message sont écrit en XML. Une fois l'entête de la requête définie et son corps ajouté, la requête est envoyée et récuperée du côté de l'application "server". La dernière étape étant de lire le contenu de la requête HTTP pour permettre son traitement.
L'envoi de telles requêtes est réalisé uniquement pour les messages ayant pour but de remplir la base de données. On retrouvera donc cette communication uniquement entre l'application "analyzer" et l'application "server", l'application métier et l'application "server" communiquent quant à eux à l'aide du dossier de communication.

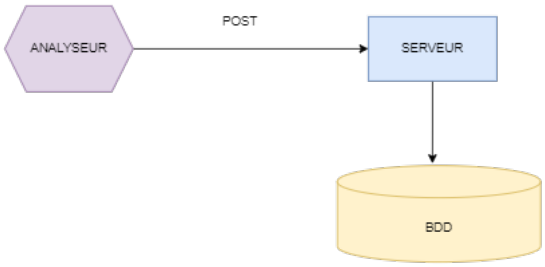

FIGURE 5 – Schéma de communication http

8.3 Code des messages

Comme énoncé plus haut dans le document, la communication est possible à l'aide de codes dans les messages. Un code message doit posséder le format $xAAA$ où x doit donner une indication sur le message, et AAA correspond à des chiffres décrivant le message.
Nous avons ainsi, les codes commençant par la lettre q qui correspondent aux messages adressant une question à l'application server, les codes commençant par la lettre r correspondent aux messages adressant une réponse au client, et les codes possédant la lettre s correspondent aux

messages indiquant à l'application *server* qu'il doit sauvegarder les données du message dans la base de données.

Les chiffres suivants la lettre donne l'indication sur le message, par exemple si le dernier chiffre n'est pas un zéro (x00A) alors le message concerne les produits, si le chiffre du milieu ne vaut pas zéro, alors le message concerne une panne, et enfin, si le premier chiffre du code ne vaut pas zéro alors le message concerne un client.

Les codes corresondant à une question sur les produits sont les suivants :

- q001 -> Recherche les produits selon leur référence et leur date de panne.
- q002 -> Recherche les produits selon leur référence et leur marque.

Le code corresondant à une question sur les pannes est le suivant : q010 -> Recherche les pannes à l'aide d'une description et du mail d'un client.

Le code correspondant à une question sur les clients est le suivant : q100 -> Recherche les clients par email, marque de produit, et référence de produit.

Comme énoncé dans la partie **7.2.2 Génération des réponses**, nous avons pu trouver un modèle commun à toutes les réponses c'est pourquoi nous avons comme unique code réponse r001.

De la même manière un message ayant pour but d'insérer des informations dans la base de donnés doit posséder le code s000.

9 Conclusion

Ce projet, dans lequel nous nous sommes investis ce semestre, nous a apporté un nombre important de connaissances en matière de documents structurés. Aucun de nous ne connaissait le langage XML il y a encore quelques mois. Nous connaissions son nom uniquement, mais c'est avec ce projet que les différents concepts de structuration des données se sont imprimés dans notre esprit, et que nous avons compris son impact dans la la gestion et l'échange d'informations. Nous sommes désormais capables d'automatiser et faciliter les échanges de contenus complexes entre différents systèmes d'information hétérogènes.